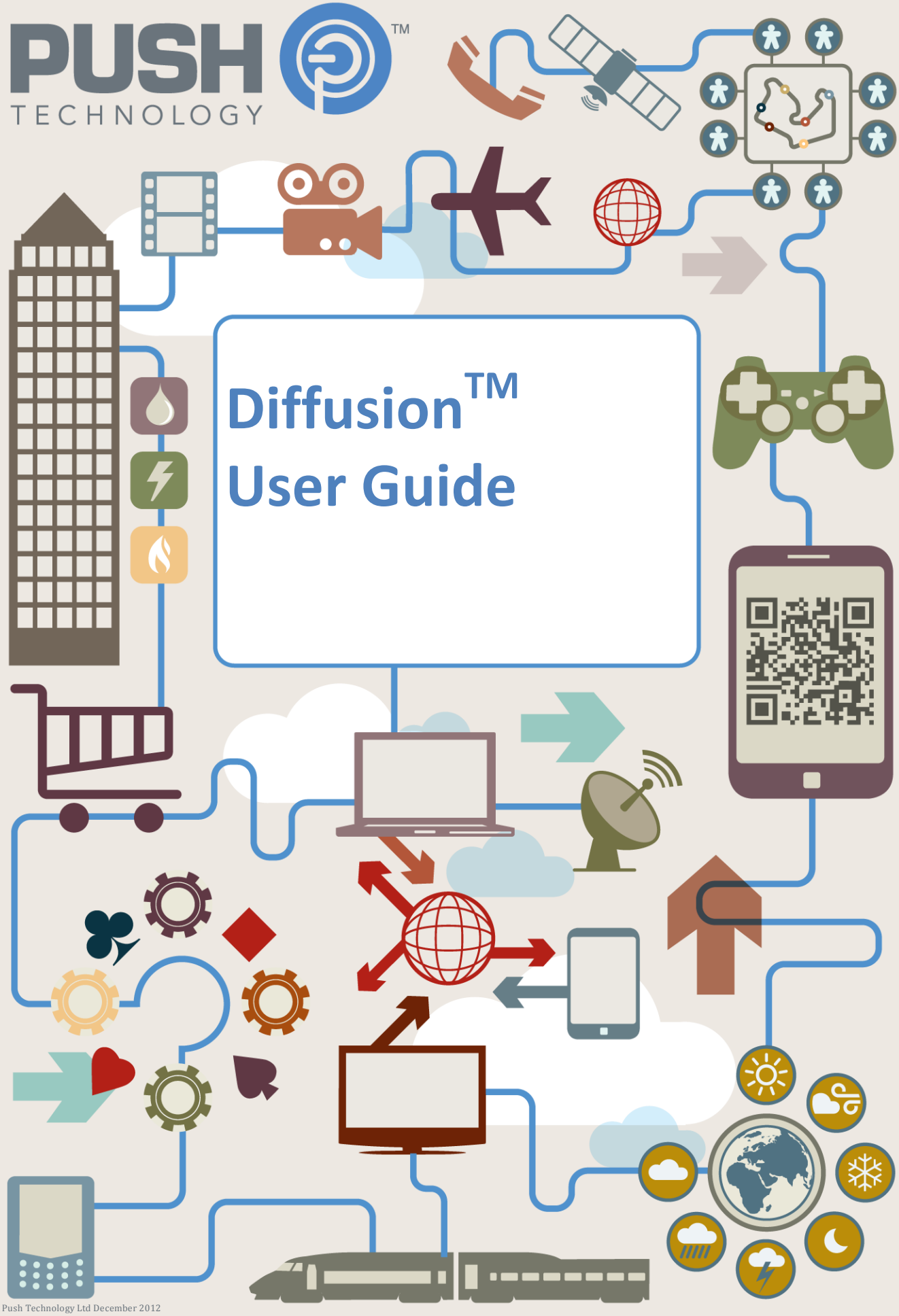
PUSH TECHNOLOGY™



Diffusion™ User Guide

©Copyright Push Technology Ltd December 2012

User Guide

Contents

List of Figures 13

List of Tables 17

Chapter 1: Welcome 21

Chapter 2: Introduction 23

Introduction 24
What's new in Diffusion 4.6? 31

Chapter 3: Quick Start Guide 33

Quick Start Guide 34

Chapter 4: Overview 59

Architecture 60
Diffusion™ Server 61
Messaging 62
Clients 63
Publishers 64
Topics 65
Message Queues 65
Concurrency 66
Glossary 67

Chapter 5: Installation 69

System Requirements 70
Graphical Installation 71
Headless Installation 71
Licensing Instructions 72
Installed Files 74
Testing your installation 75
Web Server Installation 76

Chapter 6: Server 79

Server Basics 80
Starting the Server 80
Running from Java Application 80
Concurrency 82
Connectors 85
Load Balancers 86

Chapter 7: Web Server 91

Interaction with Publishers 92

Security....92

Chapter 8: Publishers....95

Publisher Basics....96
Defining Publishers....96
Loading Publisher Code....96
Programmatically loading Publishers....97
Starting and Stopping Publishers....97
Publisher Topics....98
Receiving and Maintaining Data....100
Publishing and Sending Messages....100
Publisher Notifications....101
Client Handling....102
Publisher Properties....103
Using Concurrent Threads....103
Publisher Logging....103
Server Connections....103
General Utilities....104
Writing a Publisher....104
Creating a Publisher Class....104
Publisher Startup....105
Data State....105
Data Inputs....105
Handling Client Subscriptions....107
Publishing Messages....107
Topic Locking....108
Handling Clients....108
Publisher Closedown....109
Testing a Publisher....109
Client Queues....110
Queue Enquiries....110
Maximum Queue Depths....110
Queue Notification Thresholds....111
Tidy on Unsubscribe....111
Filtering Queued Messages....111
Client Validation....111
Client Validation Policy Types....112
Automatic or Manual Policies....112
Client Validation Criteria....112
Client Connection Validation Policy....113
Client Subscription Validation Policy....113
Using Policies Programmatically....113
Client Geo and WhoIs Information....114
The Diffusion™ WhoIs Service....115
Client Groups....116
Client Notifications....117
Adding a ClientListener....117
Using DefaultClientListener....118
Design Patterns....118
Data Models....118
Hierarchic Topics....119

Chapter 9: Clients....121

Connection....122

Topics.... 122
Messages.... 122
Client Types.... 122
External Clients.... 123
Javascript Clients.... 124
Silverlight Clients.... 124
Flash Clients.... 124
Websocket Clients.... 124
Publisher Clients.... 125
Android Clients.... 125
iOS Clients.... 125
J2ME Clients.... 125
Client Close Reason.... 125
Cross-domain.... 127
Cross Domain XML File.... 127
Flash Security Model.... 127
Silverlight Security Model.... 128
Javascript Security Model.... 128
Browsers supported.... 130
Protocols Supported.... 133

Chapter 10: Topics.... 135
Topic Basics.... 136
What is a Topic?.... 136
How are Topics Used?.... 136
Topic Hierarchies.... 137
Subscription.... 137
Topic Loading.... 137
Topic Data.... 137
Topic Attachments.... 138
Topic References.... 138
Topic Listeners.... 138
Topic Providers.... 138
Topic Hierarchy.... 138
The Topic Tree.... 140
Referencing Hierarchic Topics.... 140
Referencing Segments of the Topic Tree.... 141
Topic Naming.... 141
Topic Naming - Restricted Characters.... 141
Topic Naming Recommendations.... 142
Topic Aliasing.... 142
Creating Topics.... 143
Configuring Topics.... 143
Dynamically Adding Topics.... 143
Removing Topics.... 144
Topic Subscription.... 144
Subscription on Connection.... 145
Ad Hoc Subscription.... 145
Effect of Subscription.... 145
Subscribing to Topics that do not Exist.... 145
Subscription Using Topic Selectors.... 145
Forced Subscription.... 146
Forced Subscribe Individual Client.... 146
Forced Subscribe By Topic Set.... 147
Forced Subscribe All Clients.... 147

Pre-emptive Subscription....147
Manual Pre-emptive Subscription....147
Automatic Pre-emptive Subscription....147
Automatic Pre-emptive Subscription All Topics....148
Subscribing Clients Individually....148
Re-Subscription....148
Unsubscribing....148
Force Unsubscribing....148
Subscription Handlers....148
Topic Specific Authorisation....149
Delegated Authorisation and Deferred Subscription....149
Topic Loading....150
Subscription Processing....150
Simple Topic Loading....151
Load Message....152
Cached Load Message....152
Topic Data Subscription....152
Topic Loaders....153
Creating a Topic Loader....153
Declaring a Topic Loader....153
Mixing Simple and Topic Loader Loading....153
Topic Loaders for Multiple Topics....154
Cached Topic Loader....154
Topic Data....155
Using Topic Data....155
Publishing Topic Data....156
Paged Topic Data....169
Service Topic Data....175
Routing Topic Data....180
Remote Topic Data....183
Child List Topic Data....184
Topic Notify Topic Data....185
Metadata....188
Topic Selection....192
Topic Selectors....192
When is a Topic String a Selector?....194
Topic Fetch....194
Client Fetch....194
Server Fetch Handling....195
Topic Sets....197
Topic Attachments....198

Chapter 11: Messages....199
Introduction....200
What is a Message?....200
Message Record and Fields....200
Message Capacity....200
Maximum Message Size....201
Message Types....201
Topic Messages....201
Ping Messages....202
Acknowledged Messages....202
Control Messages....202
Creating Messages....203
Publisher Factory Methods....203

Using API Factory Methods....203
Populating Messages....203
Setting User Headers....204
Setting Data....204
User Headers....205
Adding User Header Data....205
Retrieving User Headers....205
Retrieving User Header Data by Index Number....205
Reading Messages....205
Reading Messages in the Java API....206
Encoding Considerations....206
Message Pointers....206
Reading Whole Message Content....206
Content as a String....206
Content as bytes....206
Traversing Message Content....207
Reading Bytes....207
Reading One Byte....207
Reading Array of Bytes....207
Reading Records and Fields....207
Concurrency Issues....207
Using a Message Reader....208
Records....208
Populating Messages....209
Reading Messages....210
Catering for Empty Fields and Records....211
Metadata....211
Metadata Structure....212
Message Metadata....212
Metadata Record....212
Field Metadata....212
Multiplicity....213
Examples....214
Byte Encoding....217
Set the Byte Encoding....218
Character Encoding....218
Character Encoding in the Java API....219
The Effect of Character Encoding on Message Sizing....219
Message Priority....219
Normal Priority....219
High Priority....219
Low Priority....220
Acknowledged Messages....220
Setting Messages as Requiring Acknowledgement....220
Setting Acknowledgement Timeout....221
Client Handling of Acknowledgements....221
Server Handling of Acknowledgements....221
Publisher Notification of Non Acknowledgement....222
Client Notification of Non Acknowledgement....222
Message Acknowledgement Considerations....222
Diagnostic Considerations....222
Fragmented Messages....222
Creating fragmented messages....223
Message Filters....224
Client Queue Message Filtering....224

Chapter 12: Security 227

User Access Control 228
- Authorisation Handler 228
- Authentication 229
- Authorisation 229
- Authorisation Handler 230
- Permissioning 230

Network Security 230
- Secure Clients 230
- Connector Configuration 231
- Keystores 231

Chapter 13: Distribution 233

Publisher Clients 234
- Publisher Client Capabilities 234
- Publisher Client Connection 234
- Notifications 236
- Buffer Sizes 237

Distributed Topics 237

Distribution Examples 237
- Distributors 238
- Aggregators 238
- Mixed Mode 239

Chapter 14: Event Publishers 241

Receiving Messages from Event Publishers 242

Receiving Event Publisher Notifications 242

Sending Messages to Event Publishers 242

Event Publishers Receiving Messages from Publishers 242

Configuring Event Publishers 242

Chapter 15: Remote Control 245

Remote Control 246

Remote Services 247

Remote Control Publishers 250

Chapter 16: System Management 255

General Management 256

Classic Deployment 256

Hot Deployment 256
- Deployment Methods 256
- Anatomy of a DAR file 257

Using JMX 259
- Mbeans 259
- JMX Adapter 266
- Using Visual VM 269
- Using JConsole 274

Enabling collection of Statistics 276
- Statistics 277

Diffusion Monitoring Console 277

Basic integration with Splunk 285

Chapter 17: APIs 289

Table of APIs 290
Java API 294
Client API 295
Event Publisher API 297
Remote Service API 298
.Net API 306
Client API 306
Event Publisher API 309
Remote Service API 310
Javascript API 310
Using the JavaScript API 310
ActionScript API 313
Using the ActionScript API 313
Silverlight API 316
Using the Silverlight API 316
iOS API 320
DFClient 321
Installing the docset 324
Android API 325
Using the Android API 325
J2ME API 328
Using the DiffusionClient 328
C API 331
Using the C API 331
DiffusionWrapper.js 332
How to use DiffusionWrapper.js 333

Chapter 18: Configuration 337

XML Configuration 338
server 340
connectors 355
publishers 360
web-servers 366
logs 372
management 374
statistics 375
connection-validation-policies 378
subscription-validation-policies 380
env 381
aliases 382
mimes 382
Programmatic Configuration 382
Using the Configuration API 383
The Configuration Tree 385

Chapter 19: Adapters 387

JMS Adapter 388
Installing the JMS Adapter 389
Configuring the JMS Adapter 389
JMS Adapter examples 392

Receiving messages from JMS ... 392
Sending messages to JMS ... 393
Processing a request-reply message with a Diffusion client ... 394
Sending a request-reply message from a Diffusion-client ... 394

Chapter 20: Tuning ... 397
Buffer Sizing ... 398
Message Sizing ... 399
Client Queues ... 400
Client Multiplexers ... 400
Write Selectors ... 401
Connectors ... 401
Thread Pools ... 402
Client Reconnection ... 404
Server Configuration ... 404
Message Queue Management ... 404
Client Reconnection ... 405
How to test reconnection in my environment? ... 405
Client Failover ... 406
Autofailover ... 407
Java Failover ... 407
JavaScript Failover ... 407
ActionScript Failover ... 407
Message Conflation ... 408
Conflation Overview ... 408
Conflation Business Value ... 410
How Does Conflation Work? ... 410
Configuring Conflation ... 411
Conflation Counts ... 415
Client Throttling ... 415
How Does Throttling Work? ... 415
Enabling Throttling ... 416
Memory Considerations ... 416
Garbage Collection (Java HotSpot VM) ... 416
Platform Specific Issues ... 417
Socket Issues ... 417
Increasing Number of Open Files ... 418
Publisher Design ... 419

Chapter 21: Diagnostics ... 421
Logging ... 422
Logging API ... 424
Connection Counts ... 425
Message Diagnostics ... 426
Javascript Diagnostics ... 427
Flex and Flash Diagnostics ... 431
Windows Diagnostics ... 432
Debugging a Publisher ... 433
Log Messages ... 438

Chapter 22: Introspector ... 469
Supported Platforms ... 470
Installing from update site ... 470

Installing subsequent plugin updates....473
Deinstallation....473
Finding the Diffusion™ perspective....474
Adding servers....475
Opening servers....476
Exploring the topics....477
Getting topic values....477
Configuring columns....477
Ping servers....478
Count topics....478
Using the Clients view....478
Ping....479
Statistics....479
Topics....480
Logging....480
Server logs....480
Property obfuscator....481

Chapter 23: Demos....483
Demos....484

Chapter 24: Testing....485
Flex/Flash Client....486
Java Client....489
Java Event Publisher Test Tool....493
Javascript Client Test Tool....495
Silverlight Client Test Tool....496
Stress Test Tuning....499
Stress Test....500
Test Tools....501
Windows Client Test Tool (.NET)....502
Windows Event Publisher Test Tool (.NET)....504

Chapter 25: Tools....507
Tools....508
Tools for Amazon EC2....508
Tools for Joyent....510

Chapter 26: Protocol....511
DPT....513
Connection Protocol....513
Messages Protocol....515
HTTP Protocol....520
WebSocket Protocol....524
Command Protocols....525
Service Topic Protocol....525
Paged Topic Protocol....526
Notification Topic Protocol....527

List of Figures

Figure 1: Diffusion Architecture....26
Figure 2: Diffusion Server....26
Figure 3: Scalable....26
Figure 4: Multiple Servers....27
Figure 5: Connectivity....28
Figure 6: Publisher to Client....28
Figure 7: Event Publisher to Publisher....29
Figure 8: Message Queues....30
Figure 9: Client libraries....31
Figure 10: Basic Components....60
Figure 11: Distribution....61
Figure 12: Server....62
Figure 13: Topics....65
Figure 14: Queues....66
Figure 15: Installer first steps....71
Figure 16: Finishing off....71
Figure 17: Loading a licence using the installer....73
Figure 18: Thread Diagram....85
Figure 19: Sticky-IP in F5 BIG-IP....88
Figure 20: The Message Queue....110
Figure 21: Using a load balancer to composite two URL spaces into one....129
Figure 22: Flat Structure....139
Figure 23: Hierarchical Topic Structure....139
Figure 24: Topic Aliasing....142
Figure 25: Distributors....238

Figure 26: Aggregators....239
Figure 27: Two-Tier architecture....246
Figure 28: Three Tier Architecture....247
Figure 29: The Server MBean 'stopController' operation showing in JConsole....259
Figure 30: Reflecting MBeans as topics....267
Figure 31: Showing a composite attribute as a topic nest....268
Figure 32: Topics reflecting an ArrayType and TabularType MXBean attributes....269
Figure 33: Java VisualVM....270
Figure 34: Java VisualVM monitor....271
Figure 35: Java VisualVM threads....272
Figure 36: Java VisualVM profile....273
Figure 37: JConsole....274
Figure 38: Tuning and monitoring....275
Figure 39: Monitoring a remote Diffusion server with JConsole....276
Figure 40: The default console layout....278
Figure 41: The table of Publishers....279
Figure 42: Publisher statistics graphs....279
Figure 43: The table of Topics....280
Figure 44: Details of the Topic publishing the CPU load of the host server....281
Figure 45: The table of Clients....281
Figure 46: The table of log entries....282
Figure 47: Editing the Access Policy....282
Figure 48: Notification that the Diffusion server has stopped....283
Figure 49: The default Diffusion Details panel....283
Figure 50: Editing the properties of the Diffusion Details panel....284
Figure 51: Visualising the CPU load on a server at a specific time....284
Figure 52: Editing and adding to the set of topics for this panel....285
Figure 53: XCode documentation browser....321

Figure 54: Wrapper..... 333
Figure 55: JMS adapter topic tree layout..... 388
Figure 56: Subscription flow..... 393
Figure 57: Sending flow..... 393
Figure 58: Request-reply from JMS to Diffusion..... 394
Figure 59: Request-reply from Diffusion to JMS..... 395
Figure 60: Reconnection Scenario..... 406
Figure 61: Normal and Throttled Client Queues..... 416
Figure 62: Firefox Console..... 427
Figure 63: Chrome's console..... 428
Figure 64: Internet Explorer's console..... 429
Figure 65: Opera's console..... 430
Figure 66: Safari's console..... 431
Figure 67: Example Directory Tree..... 433
Figure 68: New Java Project..... 434
Figure 69: Build Path Entries..... 435
Figure 70: Creating a new Java class..... 436
Figure 71: Example classpath entries..... 437
Figure 72: Example Arguments Tab, with licensing switches..... 438
Figure 73: Adding a Repository..... 470
Figure 74: Next step..... 471
Figure 75: Accept the license agreement..... 472
Figure 76: Click OK..... 472
Figure 77: Restarting..... 472
Figure 78: "About Eclipse" dialogue..... 473
Figure 79: Installed Plugins..... 474
Figure 80: Perspective..... 474
Figure 81: Views..... 475

Figure 82: Add a Server....476
Figure 83: Edit Server Details....476
Figure 84: View Topic Values....477
Figure 85: Re-order Columns....478
Figure 86: Ping a Server....478
Figure 87: Topic Count....478
Figure 88: Ping clients....479
Figure 89: Server log entries....480
Figure 90: Property Obfuscator Dialog....481
Figure 91: Flex Client....486
Figure 92: Sending Messages....487
Figure 93: Message tree displayed....488
Figure 94: Debugging on....489
Figure 95: External Client tester....490
Figure 96: Sending messages....491
Figure 97: Messages displayed....492
Figure 98: Examining a message....493
Figure 99: Event Publisher Test Tool....494
Figure 100: Sending messages....495
Figure 101: Javascript test tool....496
Figure 102: Silverlight test tool....497
Figure 103: Sending messages....498
Figure 104: Displayed messages....499
Figure 105: External Client test tool....502
Figure 106: Sending messages....503
Figure 107: Displayed messages....504
Figure 108: Property Obfuscator....508

List of Tables

Table 1: A Glossary of Terms..... 67
Table 2: Installed files.....74
Table 3: Tools and Utilities.....74
Table 4: Connectors properties..... 85
Table 5: Connection restrictions..... 86
Table 6: Routing strategies..... 88
Table 7: Start Publisher.....98
Table 8: Stop Publisher.....98
Table 9: Notification methods.....101
Table 10: General Publisher Utilities.....104
Table 11: Validation criteria.....112
Table 12: WhoIs.....114
Table 13: WhoIs service..... 115
Table 14: Client Listener Notifications.....117
Table 15: Client Types..... 123
Table 16: Browsers Supported.....130
Table 17: Browser Plugins.....130
Table 18: Websocket..... 131
Table 19: Cross-origin Resource Sharing (CORS).....131
Table 20: Max Connections per Domain.....132
Table 21: Restricted Characters..... 141
Table 22: Methods for Adding Topics to Publishers.....143
Table 23: Publishing Topic Data Types.....157
Table 24: Paged Topic Data Types..... 169
Table 25: Duplicates Policies.....170

Table 26: Usable Methods with Ordered or Unordered Topic Data....171
Table 27: Paged Topic Commands....173
Table 28: Paged Topic Notifications....174
Table 29: Paged Status....174
Table 30: Handling Responses....179
Table 31: Handling Errors....180
Table 32: Error Types....180
Table 33: Notification Levels....186
Table 34: Selection Modes....187
Table 35: Data types....189
Table 36: Selector Examples....194
Table 37: Message Format....200
Table 38: Separator Bytes....208
Table 39: Data Types....212
Table 40: Types of Byte Encoding....218
Table 41: Capabilities of Transports....218
Table 42: Creating a Fragmented Message....223
Table 43: Fragmented Message Lifecycle....223
Table 44: Authorisation Handler Methods....228
Table 45: Client Security....231
Table 46: Topic Types....248
Table 47: Adding Listeners....252
Table 48: Notifications as Topics....267
Table 49: Feature Matrix....291
Table 50: Java APIs....294
Table 51: Environment Properties....299
Table 52: Service Options....299
Table 53: Service Methods....300

Table 54: Service Notifications....301
Table 55: XML Value Types....338
Table 56: XML Millis Mnemonics....338
Table 57: XML Bytes Mnemonics....339
Table 58: Conflation Policy Elements....411
Table 59: Conflation Policy Modes....412
Table 60: Action depending upon merge result....413
Table 61: Targets....509
Table 62: Properties for targets start, stop and status....509
Table 63: Additional properties for targets deploy and undeploy....509
Table 64: Message Types....517
Table 65: Key....518
Table 66: Message Interactions....519
Table 67: Commands (Message Type 36)....527
Table 68: Notifications (Message Type 41)....528
Table 69: Topic Types....528
Table 70: Topic Properties....528

Chapter 1

Welcome

Welcome to Push Technology's User Manual for Diffusion™

The manual is regularly updated, but if you require further help, please contact the team at support@pushtechnology.com

Copyright 2013 Push Technology

Chapter 2

Introduction

Topics:

- *Introduction*
- *What's new in Diffusion 4.6?*

About Diffusion

Diffusion is a unique integration of messaging with in-memory data caching and in-flight data processing allowing consistent, current data to be delivered to tens of thousands of client devices with optimal resource utilisation.

Introduction

An Introduction to Diffusion™

Diffusion™ is a unique integration of messaging with in-memory data caching and in-flight data processing allowing consistent, current data to be delivered to tens of thousands of client devices with optimal resource utilisation.

Diffusion™ streams high frequency volatile data efficiently, with minimal overhead under normal operating conditions, with smart recovery when connectivity between services and clients fail due to loss of connectivity or changing bandwidth and latency conditions.

Diffusion™ conserves bandwidth while increasing application performance with negligible sub-100 microsecond latency to the web and mobile cellular clients typically experiencing 10s to 100s of milliseconds.

Diffusion™ multiplexes data and media streams, pushing them over a wide selection of transport protocols to internal and internet facing clients in such a way that a single data distribution service can service low latency internal clients and low fidelity external clients with a single set of deployed services.

Diffusion™ is smart, leveraging data types and structures and service qualities allowing guaranteed delivery with acknowledgement and non-durable loss detection and recovery.

Diffusion™ performs structural conflation, ensuring current and consistent replace (latest value, no stale data) and merge (current, and consistent based on user defined logic) in soft real time.

Diffusion™ can fragment low priority data and interleave it with high priority data to maximize bandwidth utilisation and minimise delays in content delivery.

- Real-time dissemination of volatile data over the Internet.
- Gathering, collating and publishing data from multiple sources before distributing over a network.
- Native online applications that operate beyond traditional web browsers using native or web based transports.
- Highly scalable low-latency applications where enormous amounts of data move back and forth.
- Transportation applications where delivery of the right information to the right person at the most critical time is paramount - no matter what the end-device.
- Ease of integration with enterprise messaging which support durable store-and-forward messaging, enabling the guarantees of traditional message oriented middleware to be optimised for delivery over medium to low fidelity networks with optimal bandwidth utilisation.

Diffusion™ is versatile. User-defined publication logic can be deployed in the system allowing bespoke routing, computations and delivery algorithms to be easily composed with minimal additional latency.

Diffusion™ is scalable. Diffusion™ can be deployed behind hardware and software load balancers for typical web-based edge of the network deployments and configured in networks of distributing, aggregating and pipelined architectures with broker-based or brokerless architectures for cooperative and collaborative clustered data distribution and analytics behind the edge.

Diffusion™ is smart. The Diffusion™ protocol deviates from traditional messaging styles, optimising for bidirectional streaming conversational interactions between machine generated data and human participants, without compromising pedigree of machine to machine interactions.

Diffusion™ is a cache. Diffusion™ can provide both stateless and stateful topics. Stateful topics enable efficient recovery where clients are likely to fail frequently for short periods. Diffusion™ can issue a snapshot of the current state on reconnection, but typically pushes changes or deltas under normal operating conditions. Similarly, Diffusion™ clients can push snapshots of data that has changed from their perspective to diffusion services on reconnect.

Diffusion™ Data Distribution Engine

Provides machine-to-machine and machine-to-human messaging within an intranet/extranet environment.

Features

Copyright 2013 Push Technology

A highly efficient low latency messaging server that offers high performance due to its ability to provide in excess of 1,000,000 1000-byte messages per second using a single instance on an industry-standard server.

Diffusion™ can distributed in excess of 5,000,000 125 byte messages per second on the same commodity hardware with a single 10-gigabit ethernet interface. Increased efficiency is achieved through a packed-binary wire format and distributing full state only on subscription to a topic with deltas or changes subsequently. The engine can be deployed as a standalone instance or in highly and easily scalable clusters behind hardware of software-based load balancers to linearly scale systems to meet the demands of 100s of 1000s of concurrently connected streaming clients. This gives organisations the opportunity to scale up operations with zero fan-out.

- **Scaling with minimal latency.** No stale data for conflatable topics, and complete consistency where merge or replace based structural conflation is utilised.
- **Geo-location of data.** Keeping data efficiently synchronised and current across multiple geographic locations, allowing teams to focus on local access and leverage of remotely distributed data sources.
- **Future-proofing enterprise messaging and communications.** Providing the ability to merge and distribute data from multiple sources across an advanced highly scalable distribution platform

Diffusion™ Data Distribution over the Internet

delivers minimal latency real-time current and consistent data to any net-connected device including web browsers, mobile phones, third party applications and media platforms such as Internet TVs and game consoles.

Features

- **Intelligent message delivery** guarantees service and manageability of the data directly to the end-user.
- Bi-directional support for messages/events ensures speed is consistently maintained both to and from the end user.
- Cascading Transport ensures the fastest possible message delivery and graceful degradation to support the latest web-based standards and legacy browsers via Ajax/Comet and IFrame based connectivity in the browser.
- High performing engine ensures that data is distributed with minimal latency and minimal infrastructure outlay reducing total cost of operations.
- **Extensible type system** enables any type of content to be used.
- **Zero fan out** enables data to be sent to multiple users and channels simultaneously.
- **Easy to use API** ensures rapid development, deployment and integration with a single distribution platform for data delivery inside and outside the firewall and between remote geographic locations.

Environment Overview

Scalable Distribution

Diffusion™ is component-based and modular. Typical components are:

- **Diffusion™ Server** A highly efficient and low-latency messaging server.
- **Publisher(s)** A Publisher is a user-defined object deployed within a Diffusion™ Server, providing one or more Topics on which it publishes Messages. One or more Publishers may be deployed with a Diffusion™ Server.
- **Remote Services**. A Remote Service can perform many of the tasks of a Publisher from outside a Diffusion™ Server and feed data through the Diffusion™ Server to Clients.
- **Event Publishers** Applications that are able to connect to a Diffusion™ Server and publish Topic Messages. These Messages are routed to the Publishers that own those Topics. The purpose of an Event Publisher is to feed data into a Diffusion™ Server and thus into Publishers.

One of the benefits of component-based architecture is the ability to build your architecture as your business requirements grow. Some Diffusion™ components are optional and not necessarily required to create a working platform, only the Diffusion™ Server, Publisher(s) and Client(s) are essential components. Diffusion™ supports global distributed architectures with the ability to handle tens of thousands of concurrent connections.

Copyright 2013 Push Technology

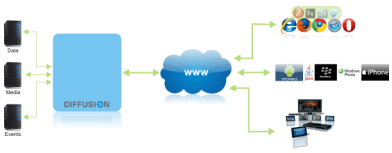


Figure 1: Diffusion Architecture

Technical Overview

Diffusion™ Server

Diffusion™ Server is a highly efficient low-latency distribution engine capable of communicating via multiple protocols, data caching and state maintenance.

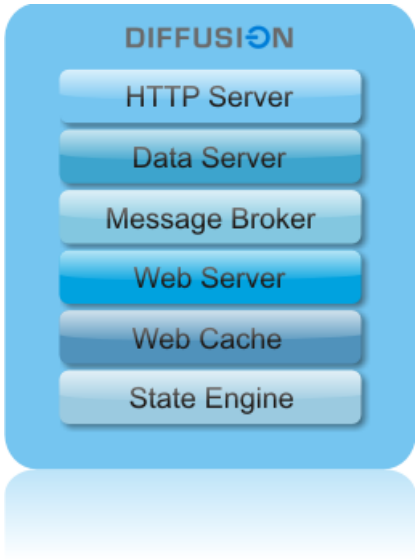


Figure 2: Diffusion Server

Diffusion™'s multiple components guarantee maximum performance with great compatibility. One server can handle all message streams or several servers can be used, each handling one message stream. The Diffusion™ Server provides the environment in which one or more Publishers can run.

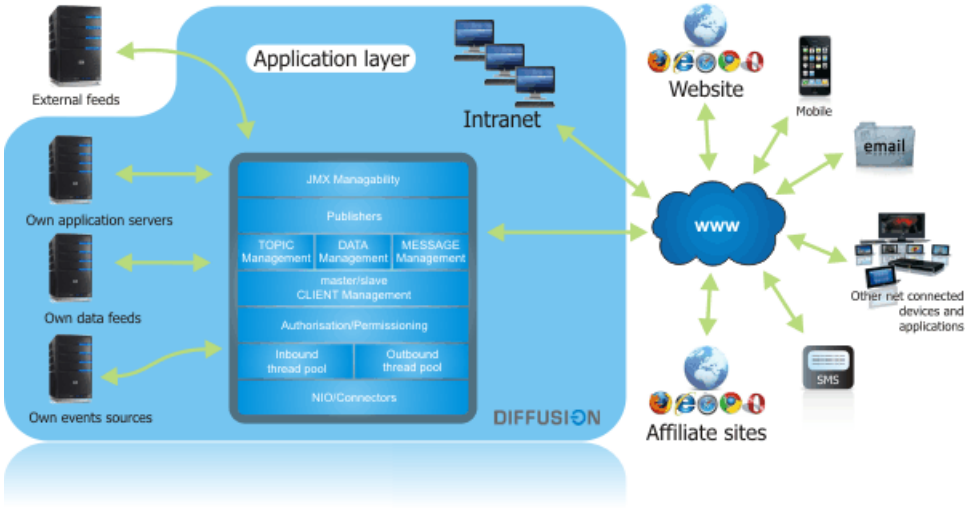


Figure 3: Scalable

Copyright 2013 Push Technology

A single Diffusion™ server may be deployed to push messages to web browsers (or other net devices) or many interconnected Diffusion™ servers can provide a fully scalable enterprise messaging solution. Multiple Diffusion™ servers can interact with clients or other servers via load balancers and firewalls. Diffusion™ is deployable as a broker based standalone service, within a collaborative distributed cluster, or broker-less embedded into JVM-based application server environments.

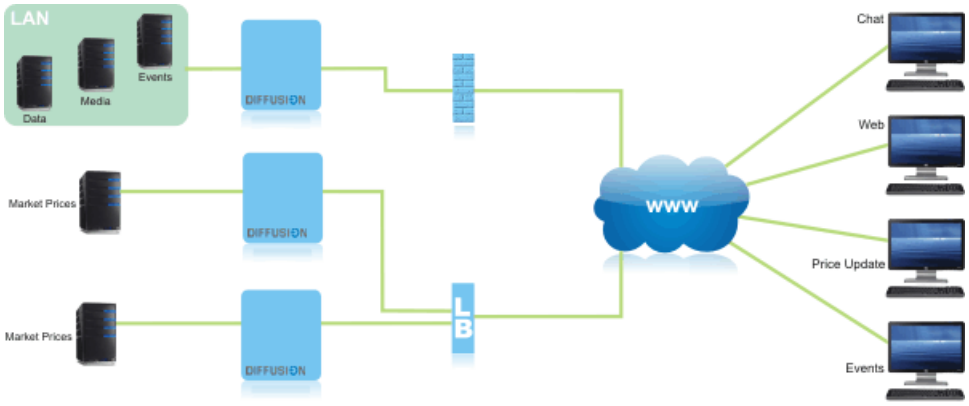


Figure 4: Multiple Servers

Distribution

Diffusion™ Servers can communicate in a peer-to-peer network to provide fully scalable distribution. Load may be spread across different Publishers on multiple Servers. Publishers may act as Clients of other Servers by subscribing to topics in the same way as normal Clients.

Diffusion™ Connectivity

Clients connect to Diffusion™ via Connectors. To provide a high level of control over connections it is possible to have any number of Client Connectors each listening on a different port. A Client Connector can act as a gateway allowing many different types of Client (HTTP, Browser, Plugin, API etc) to connect to a single connection point listening on any port you choose. A connector can also understand many protocols at the same time, for example, SSL and DPT. There are also Connectors that provide connection points for Event Publishers and/or for serving policy file requests. A Connector can be controlled via a connection policy, so only connections that are allowed can connect. The Diffusion™ Server has both Web Server and Media Server components. The Web Server is able to serve file requests as expected as well as handle Web Socket connections. The Web Server will minify the HTML, compress it and then serve the HTML to the client, so that minimal bytes are put on the wire and can also handle progressive download for multimedia content. Thread Pools are used for processing incoming Messages so that a thread is not dedicated to a single connection. Publishers may also use thread Pools for background tasks. Publishers publish (or send) messages onto Client queues where Client Multiplexer threads dispatch them onwards to the clients.

Copyright 2013 Push Technology

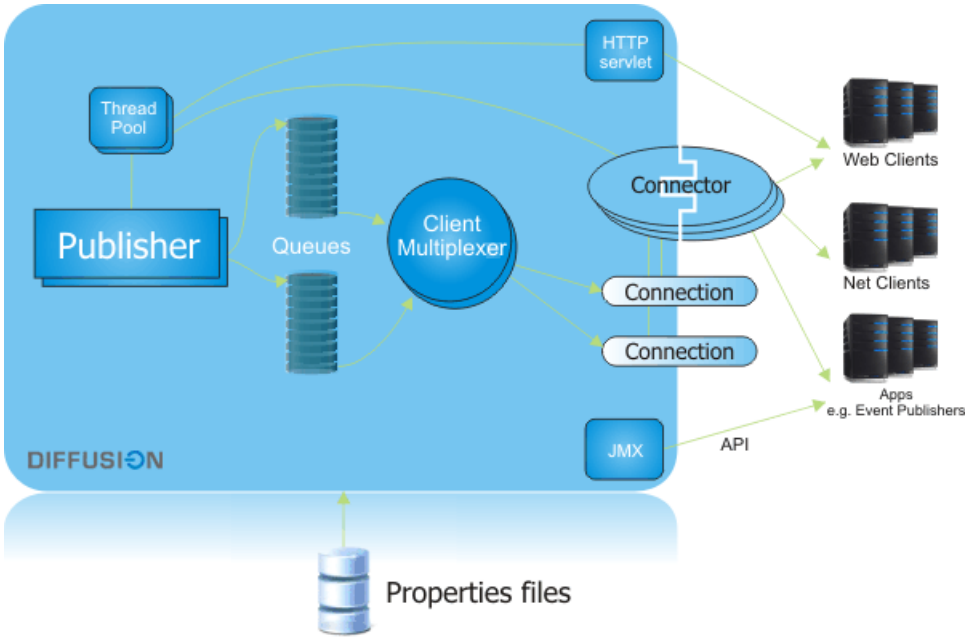


Figure 5: Connectivity

Publishers

Publishers are the core of a Diffusion™ Server. They create and maintain topics, to which external clients subscribe, and coordinate the distribution of information throughout the time the subscription remains active. Publishers are typically written in Java and deployed within the Diffusion™ Server but it is also possible to deploy an External Publisher outside the Diffusion™ Server environment. The Diffusion™ publisher requires minimum configuration to function so implementation can be as simple or complex as your business solution requires. A Publisher must maintain its own data model. The Publisher can initialise its data at start-up and then update as a result of external events. Data updates are received as appropriate, but Diffusion™ provides the Event Publisher facility which can feed messages directly into Publishers.

Publisher to Client

When a Client first subscribes to a topic the Publisher can provide the Client with the initial (or current) state of the data relating to that topic. This is called 'Initial Topic Load'. A Client may fetch the state of any topic (even if not subscribed) and the current state of the topic will be returned. A Publisher will maintain any changes to its topic data state and publish those changes to the topic as 'delta' messages.

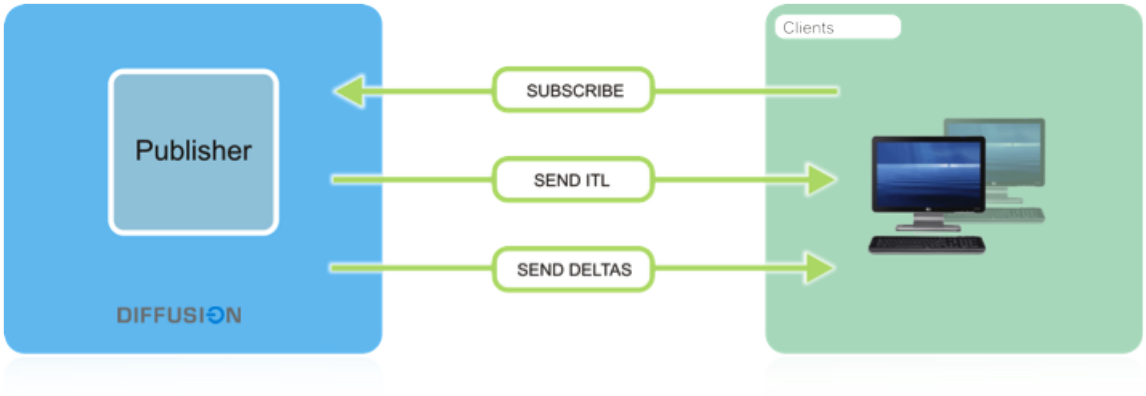


Figure 6: Publisher to Client

Copyright 2013 Push Technology

The Publisher can be configured for the ability to send messages to groups of Clients or the option to publish a message to all but a specified Client. This can be useful when a message is published in response to a message from that Client.

Event Publisher to Publisher

An Event Publisher may send Topic Messages to a Diffusion™ Server, which will be routed to the Publisher of that Topic. The Event Publisher API is a bi-directional communications channel, which is normally implemented in a third party system.

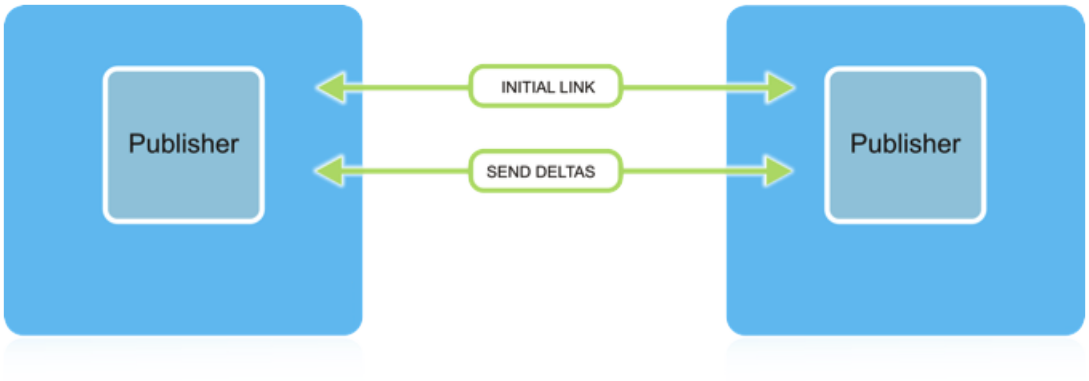


Figure 7: Event Publisher to Publisher

Remote Service to Publisher

A Remote Service can connect to to a Diffusion™ Server and establish it's own Topic 'domain' at the Server. It can create Topics and publish Messages via those Topics. It can authenticate Client connections and subscriptions.

Topics and Messages

A Topic is a named object that is registered (or added) in a Diffusion™ Server by a Publisher and provides not only basic publish/subscribe messaging but also a number of other features:

- Publishing to all subscribed Clients
- Publishing to all but one subscribed Clients
- Sending messages to individual clients
- Client to Publisher communication
- Client fetch
- Receiving Event Data
- Distributing Publishers

Publisher to Client messaging

The following messaging options exist from Publishers to Clients, and may also be 'expedited' or 'requiring acknowledgement':

- Publisher publishes message(s) to all Clients subscribed to a topic.
- Publisher publishes message(s) to all Clients subscribed to a topic except a specified Client.
- Publisher sends message(s) to individual Client.
- Publisher sends message(s) to a defined group of Clients.
- Publisher sends message(s) containing current state of topic to Client in response to a 'fetch' request (see below).

Client to Publisher

A Client may send a message via a topic that the Server will route back to the Publisher of that topic. Such messages may be set as 'requiring acknowledgement'. A Client can send a request for the current state of a topic (fetch), even topics to which it is not subscribed.

Message Queues

Copyright 2013 Push Technology

Publishers Publish messages to topics and all Clients subscribed to those topics will receive the messages. Publishers may also send messages to individual Clients or logical groups of Clients. Clients may send messages to topics, which are routed back to the topic's Publisher. Every Client has its own message queue within the Server. When a Publisher publishes a topic message it is put on the queue of each Client subscribed to that topic. The same applies to individual messages to Clients. Messages nmay be queued at Normal, High or Low priority. Conflation is used to prevent message duplication and thus reduce message traffic. Structural conflation allows messages queued for the same Topic to be merged into a single Message. Throttling is used to reduce the rate at which messages are de-queued and sent to Clients. Queue sizes can be limited so that they do not grow indefinitely. A Publisher can be notified when a queue size reaches pre-defined thresholds. The diagram shows 4 messages being created by 2 Publishers on two different topics. The message numbers indicate the order in which the Publishers publish them. Clients 1 and 2 are subscribed to both topics and receive all 4 messages. Client 3 is subscribed only to topic B and only receives messages from Publisher 2.

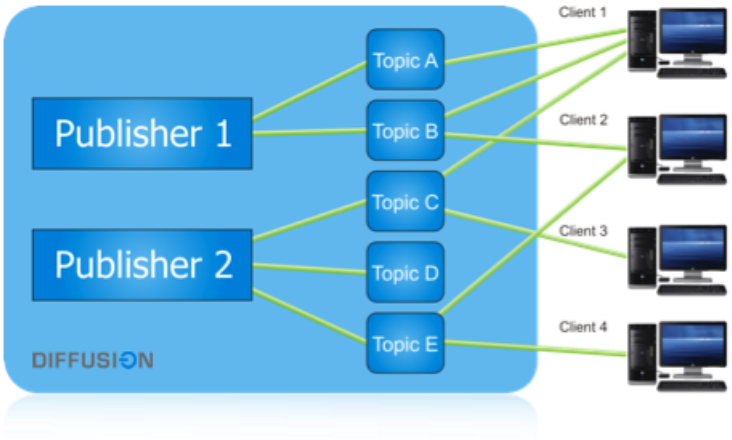


Figure 8: Message Queues

Clients

A Client is defined as an application communicating with a Diffusion™ Server using a Diffusion™ Client protocol. Because it uses the Java NIO Framework, Diffusion™ can support a very large number of concurrently connected Clients. A Diffusion™ Server hosts Publishers that push (or publish) messages to Clients. A Publisher may send messages directly to a Client (or group of Clients) or may publish messages to all Clients that are subscribed to a particular topic. Diffusion™ provides client libraries for a number of different platforms that enable applications in many different environments to act as Diffusion™ Clients. This extensive list of libraries is complimented with the ability to communicate via Websockets, Flash, Silverlight and other transports.

Copyright 2013 Push Technology

Figure 9: Client libraries

Web Clients

A web client can be a browser application using one of the various Diffusion™ interfaces: HTTP Comet, HTTP Callback or IFrame, Flash, Websockets and Silverlight.

Enterprise Clients

A Client can be any application connecting to Diffusion™ over a TCP connection, which can be over the Internet or an Intranet/Extranet. Java and .NET external client APIs are available. Other languages are also supported using the raw Diffusion™ Client Protocol, which we will refer to as DPT, and the C API suitable for embedded devices and microcontroller based devices. Mobile Clients APIs are available for mobile devices: J2ME, iOS, Blackberry, Android and Windows phone.

Publisher Clients

A Publisher can effectively become a Client when it subscribes to a topic on another Diffusion™ Server. This provides for a distributed Diffusion™ system.

What's new in Diffusion 4.6?

The latest version of Diffusion contains several new functionalities plus performance enhancements and bug fixes.

Key Features

A complete list of Diffusion's latest updates can be found in the Release Notes at *Download Diffusion*

New Monitoring Console

It is now possible to monitor Diffusion Server, Topics, Clients and MBeans using the Diffusion graphical user interface or an existing proprietary monitoring application such as Splunk™. You can expose many critical Diffusion performance statistics such as client connections, message and system performance. In fact, any metric that is exposed via JMX can also be monitored such as CPU usage, network card health or system information. JMX compatibility information is available in the Diffusion User Guide.

Using the Diffusion Console you can create your own dashboard to monitor specific user defined publishers or clients enabling you to show the information that is critical to you and your business.

Another new feature is enabling Diffusion metrics run-time configuration, so you can apply business-logic around the information your monitoring tools see.

Copyright 2013 Push Technology

Logging Improvements

We have defined clearer log file messages to make support of Diffusion even easier. Log file entries have been standardised to provide easier referencing in our Diffusion User Guide.

Ordered Paged Topic Data

Diffusion can now provide Paged Topic Data in a specific order providing greater flexibility in the way you publish data.

Protocol Buffers Topic Data – Field Removal Handling

Push Technology have enhanced the PB Topic Data for better handling of null or empty fields.

Greater Support of Topic types by Remote Control

The list of Topic types now available to Remote Control has increased so you can use Slave, Protocol Buffers, Topic Notify and Child List Topic data.

Topic Notification

To facilitate the greater monitoring features in Diffusion we have now included Topic removal or update notifications to our Topic data.

Batching of Fetch Replies

It is currently possible to request a fetch of many Topics using a Topic Selector. In some cases this can result in a large number of fetch reply messages being sent to the client at once, which can cause the client queue to fill and disconnect the client. By batching fetch replies you can now specify a mechanism where large numbers of fetch replies can be sent in batches with a time period in between.

Programmatically loading of Publishers

A Publisher can now be created from within your application.

Copyright 2013 Push Technology

Chapter 3

Quick Start Guide

Topics:

- *Quick Start Guide*

This starter guide will help you understand the fundamentals of Diffusion. It will help you to create an application that accepts data from an external data source and sends messages to an instance of Diffusion . Using Diffusion, you will be then able to publish the data to any clients that are connected and have shown an interest in specific data. This guide is designed to provide you with an insight into the power of Diffusion and give an understanding of how Diffusion can be integrated into your own architecture.

Quick Start Guide

Before you begin with the development of an application using Diffusion there are a number of pre-requisites to building a Diffusion application. This section will guide you through these requirements and at the end of the process you will be ready to start your development.

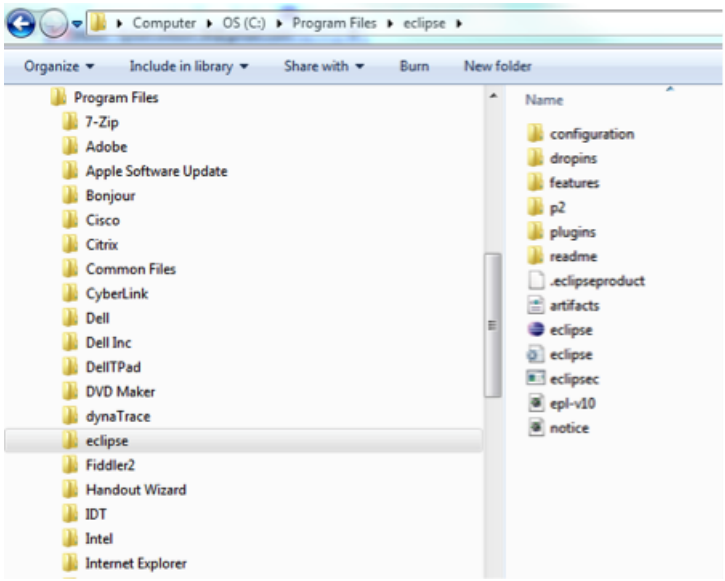
You will need to ensure you have a recent version of the Eclipse IDE installed locally, you have downloaded and installed Diffusion locally and you have installed the Eclipse plugin that is available with Diffusion. The reference materials that can be utilised during this development process include:

Online API documentation that is located via the following URL

Download Diffusion

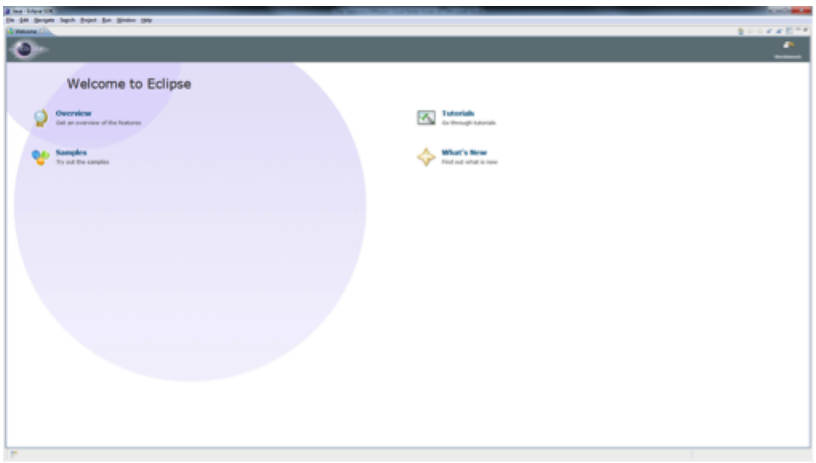
These documents include the complete Diffusion developer's manual and API documentation for all the client libraries. Eclipse Installation. Before you start developing a new application using Diffusion you require an Eclipse development environment installed on your local development machine. If you do not have Eclipse installed, you can navigate to the URL below and download the latest version. Simply follow the instructions of the download; it will prompt you to save or open the Eclipse .zip file. http://www.eclipse.org/downloads/ Ensure that when selecting the version of Eclipse to download, you specify the machine and operating system you are going to be running Eclipse within. You can use other Java development tools, but Diffusion's Eclipse tooling makes it easy to create an application. We recommend you use Eclipse until you are familiar with the structure of a Diffusion application.

Save the file to a directory that you will be able to reference at a later date. Once you have downloaded the file, you can unpack the contents into an Eclipse folder on your local machine:



Copyright 2013 Push Technology

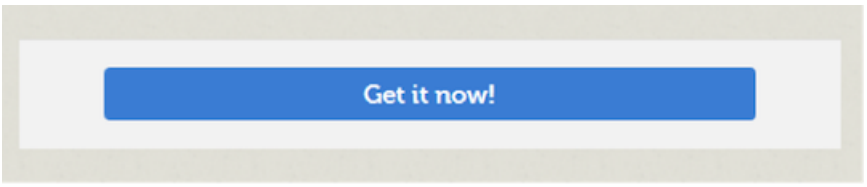
Within the folder structure of the unpacked application you will find the eclipse.exe file available in the top level Eclipse folder. By double clicking the eclipse.exe file you will initiate the Eclipse program and you are ready to start your development.



You have successfully downloaded and installed the Eclipse IDE. You are now ready to move onto the next stage.

Diffusion™ Installation

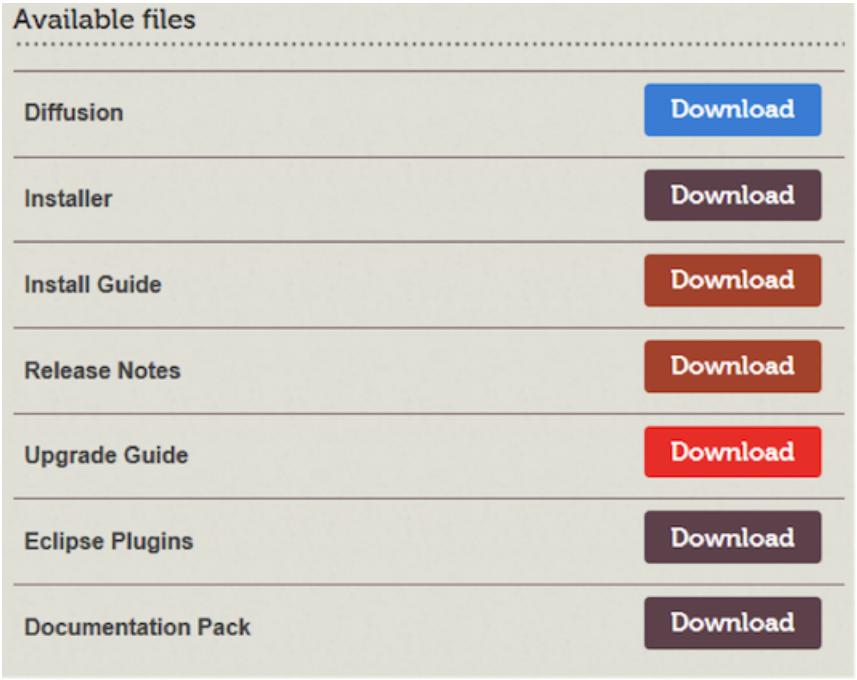
Diffusion can be installed either through the graphical installer provided or, where appropriate, using terminal commands (headless mode).There are three files that will need to be downloaded from the following URL:http://download.pushtechnology.com/releases Click on the ‘Get it now!’ option.



The three files required are:

- Diffusion(Diffusion (version_id)).jar
- Installer (install.jar)
- Eclipse Plugins (Introspector(version_id)).jar

Copyright 2013 Push Technology



Download these jar files into a temp directory on the server you want to install Diffusion on and decide which installation method you want to use, graphical or headless.

Diffusion™ Licence File

You will also need to request a Diffusion licence file from Push Technology. Diffusion will not start if it does not recognise a valid licence file and you can request a trial licence from:

http://download.pushtechnology.com

By selecting the option to request a free trial licence you will be asked to provide contact details and specify what version of Diffusion you are going to be using. Once Push Technology has received your request we will email the trial licence file to the email address you have specified.

Graphical Install

To install Diffusion using the graphical installer, navigate to the folder where you saved the three files you downloaded and double click the install.jar. The graphical installer can also be launched from the command line by typing the following:

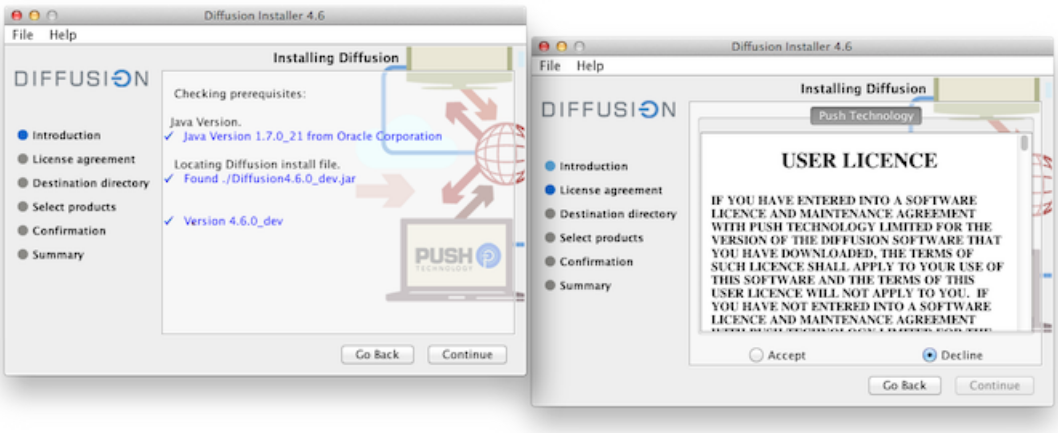
```
java -jar install.jar
```

The Install wizard starts, guiding you through the Diffusion installation process.

The installer first checks software and system requirements are satisfied for Diffusion to run. Specifically, it checks for the presence of a compliant JDK or JRE version as well as a suitable Diffusion jar file. You will be prompted to accept the End User License Agreement (EULA) and to enter the installation path for the Diffusion files. The recommendation is to select all files available within the Diffusion product. Select ‘All’ when you are presented with step 4. The installer will install the Diffusion product suite into the directory specified.

Steps 1 and 2 below:

Copyright 2013 Push Technology

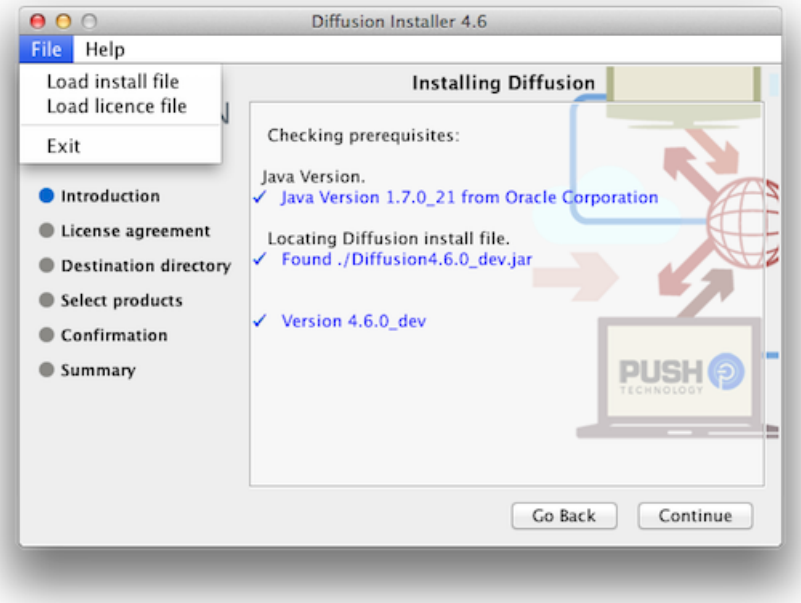

Diffusion Installer 4.6
File Help
Installing Diffusion
DIFFUSION
Introduction
License agreement
Destination directory
Select products
Confirmation
Summary
Checking prerequisites:
Java Version.
Java Version 1.7.0_21 from Oracle Corporation
Locating Diffusion install file.
Found ./Diffusion4.6.0_dev.jar
Version 4.6.0_dev
Go Back
Continue
Push Technology
USER LICENCE
IF YOU HAVE ENTERED INTO A SOFTWARE LICENCE AND MAINTENANCE AGREEMENT WITH PUSH TECHNOLOGY LIMITED FOR THE VERSION OF THE DIFFUSION SOFTWARE THAT YOU HAVE DOWNLOADED, THE TERMS OF SUCH LICENCE SHALL APPLY TO YOUR USE OF THIS SOFTWARE AND THE TERMS OF THIS USER LICENCE WILL NOT APPLY TO YOU. IF YOU HAVE NOT ENTERED INTO A SOFTWARE LICENCE AND MAINTENANCE AGREEMENT
Accept
Decline

Steps three and four - select the install destination and all the components you want to install. The recommendation is to create a Diffusion directory on your local machine and select all the components for installation:

The final steps are for confirmation of the components you are going to install and completion of the install process:

During the installation process you also have the ability to install the licence file you obtained earlier. Once requested, the licence will be emailed to you by Push Technology. If you copy the file to a directory on your local machine you will be able to reference the file during the installation process. To enable this feature, select the Load licence file option from the File menu and load the file from the directory you copied the licence.lic into.

Loading a licence file:



Copyright 2013 Push Technology

Headless Install

Installation in headless mode may be used in circumstances where the (graphical) installer cannot be used or is not appropriate. To install Diffusion in a headless environment open a terminal window, change the directory to that where the Diffusion jar files are located and type the following:

```
java -jar install.jar Diffusion(version).jar
```

The headless installation process follows the same process as the graphical mode, with the exception that the interaction takes place via the terminal rather than a graphical widow.

Similar to the graphical installation, you will be prompted for acceptance of EULA and the installation path. During the installation you will be asked if all packages need to be installed. The recommendation is to agree to this as it might cause problems further down the line if you have not installed the entire Product suite.

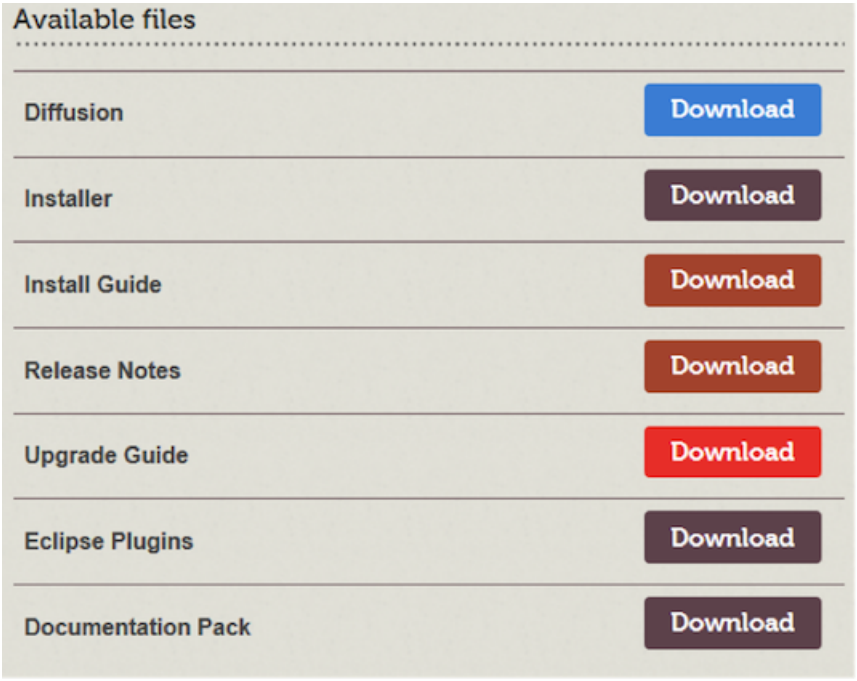
The rest of the installation will follow the same process as the graphical installer. If you are using the headless install, you will need to copy the licence.lic file directly into the etc folder before Diffusion starts.

Eclipse Plugin Installation

The Diffusion plugin available for Eclipse can be obtained from the following URL:

http://download.pushtechnology.com

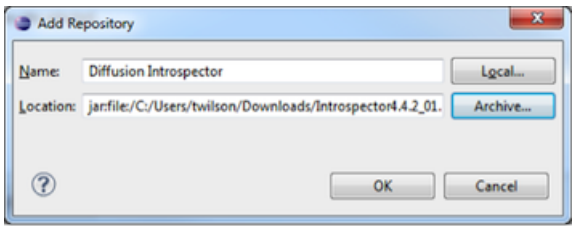
Download the Eclipse plugin onto your local machine in a directory you will be able to reference at a later date. Accept the Licence agreement to complete the download.



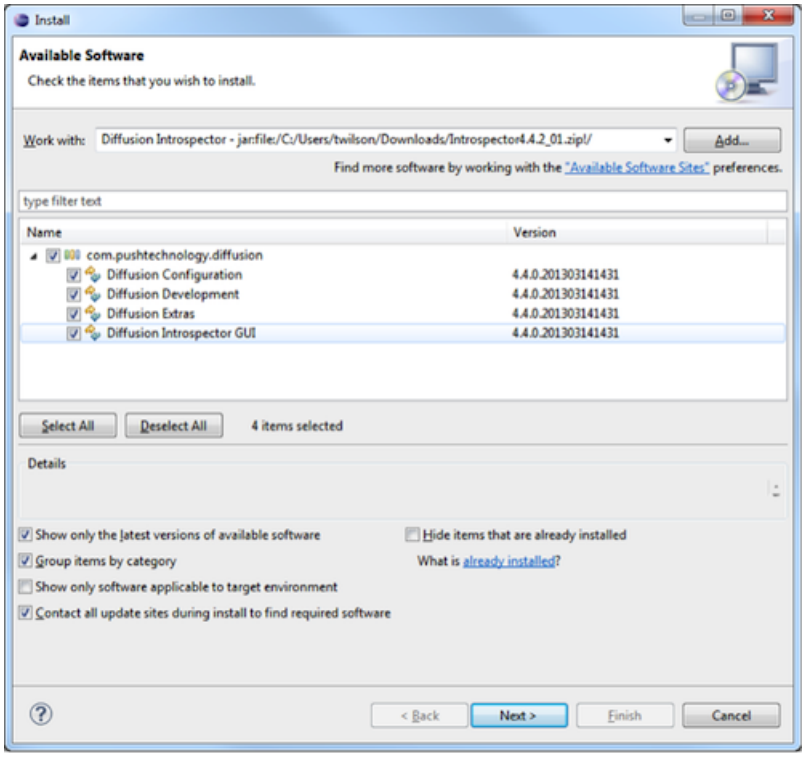
Follow the steps shown below to install the software within the Eclipse development environment.

- Select the 'Help' menu, click the 'Install new software' menu item.
- From the dialog box, choose the 'Add ...' button.
- Enter a name for the Diffusion plugin (typically Diffusion Introspector) and then select the Archive button to navigate to the Introspector jar file that you downloaded from the download web page mentioned above, then click OK

Copyright 2013 Push Technology



Select all the features that are provided by the plugin:



For future reference if you want to check if there are any updates to the Eclipse plugin then click on the help menu and select “Check for Updates”.

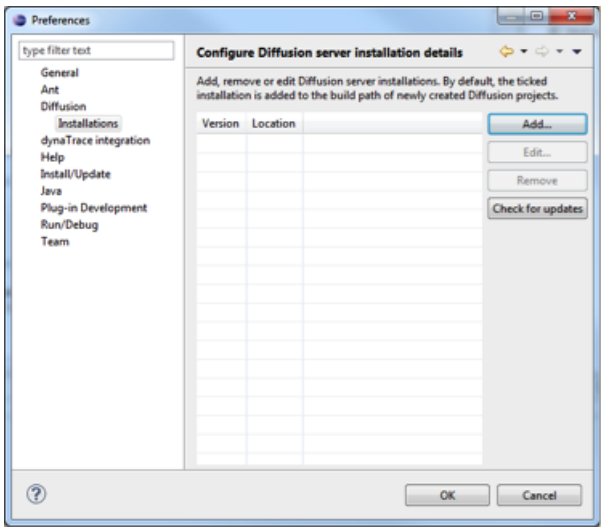
Eclipse Configuration for Diffusion

The final stage of the installation process is to configure a reference property within Eclipse that allows Eclipse to understand that Diffusion has been installed and can allow development and deployment of the Diffusion product within a java application.

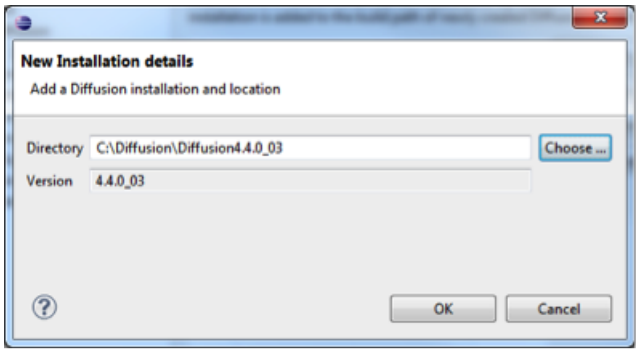
You will need to create a reference property for the Diffusion installation on your local machine. The steps mentioned below guide you through this process.

Within the Eclipse IDE, open the Window file menu and select Preferences. On the left side of the window displayed you will notice a reference to Diffusion. Expand the Diffusion tree and highlight Installations.

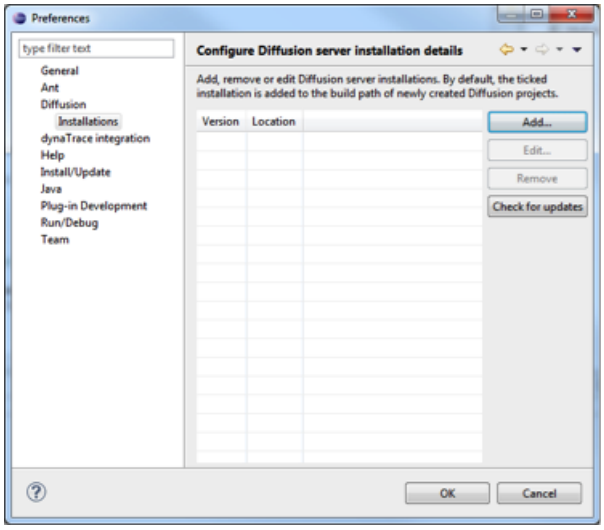
Copyright 2013 Push Technology



Select the Add option on the right side and navigate to the local installation of Diffusion.



Once you have selected OK, the previous screen will display reference to the installed Diffusion product. This will allow Eclipse to reference the Diffusion product and when you begin your development Eclipse will compile the source code with reference to the Diffusion product.



You are now ready to build an application using Diffusion.

Copyright 2013 Push Technology

Building a Diffusion™ Application

You have now completed all the preparation tasks required to provide a development environment that can be used to build and deploy a Diffusion application. This section provides you with knowledge, examples and guidance on how to build a Diffusion publisher that can be deployed within a 'cloud' environment or within your own data centres.

Overview

This guide is designed to give you a basic understanding about Diffusion and its main features and components. You are going to be building a simple Diffusion application that will accept data from an external source and publish it via a Diffusion server in the form of a publisher. Now that the development environment is setup we can move on and build a Diffusion publisher. The Diffusion plugin that you previously installed into Eclipse provides you with a standard Diffusion publisher and we will use this as a basis for your application.

The Diffusion™ Publisher

Before you begin the development of a Diffusion publisher lets recap on some of the fundamental concepts of a publisher and get an understanding of how it needs to be implemented.

- A Publisher is a user defined object deployed within a Diffusion server that provides one or more topics on which it publishes messages to clients. There may be one or more publishers deployed with a Diffusion server. Clients connect to the server and subscribe to topics.
- Every publisher that operates within a Diffusion server must be specified in the Publishers.xml configuration file, unless "HOT" deployment has been used (see section on "Deploying a Publisher" for further information).
- When a publisher starts up there are a number of steps that the publisher performs. The Publishers.XML file is found in the /etc directory and is one of the first files that is read when the Diffusion server starts.
- During this process the publisher can add any initial topics that have been specified. It can create a server connection to any severs that have been defined and once the publisher has started it will execute a method called initialLoad().

There are a number of call-back methods that form the logic of a Diffusion publisher. You will need to enhance these methods to provide more specific functionality for your own use case. The main call-back methods are:

- `initialLoad()` - This method is called when the publisher starts, typically preparing the publisher for processing.
- `subscription()` - This method is called when a client wants to subscribe to a topic (s). References to the topic and the client are passed to the method.
- `unsubscription()` - This method is called when a client unsubscribes from a topic that is owned by the publisher. References to the topic and the client are passed to the method.
- `messageFromClient()` - This method is called when a message is received from a client. References to the message and the client are passed.
- `messageFromServer()` - This method is called when a message is received from a server connection. References to the message and the server are passed.
- `messageFromEventPublisher()` - This method is called when a message is received from an event publisher. References to the message and the event publisher are passed.

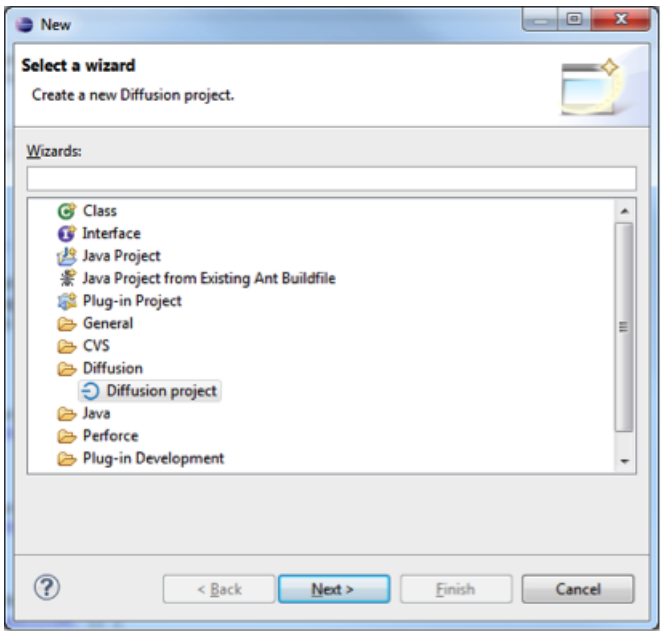
Building a Publisher

You are now going to create a Diffusion publisher by using the standard 'Echo' publisher that is provided with the Diffusion plugin. Once you have created a Diffusion project within the Eclipse development environment you will be able to write a more customised application, specific to your own use case.

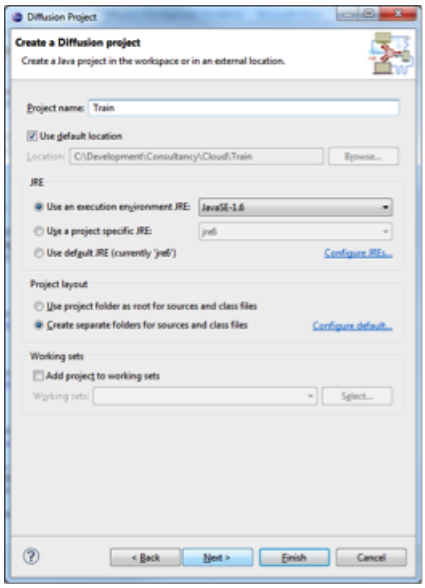
The following steps will allow you to create a new Diffusion specific project:

1. Within Eclipse, open the File menu option in the top left corner and select File / New / Other. In the window provided navigate to the Diffusion reference and expand the option. You will see Diffusion project is available.

Copyright 2013 Push Technology

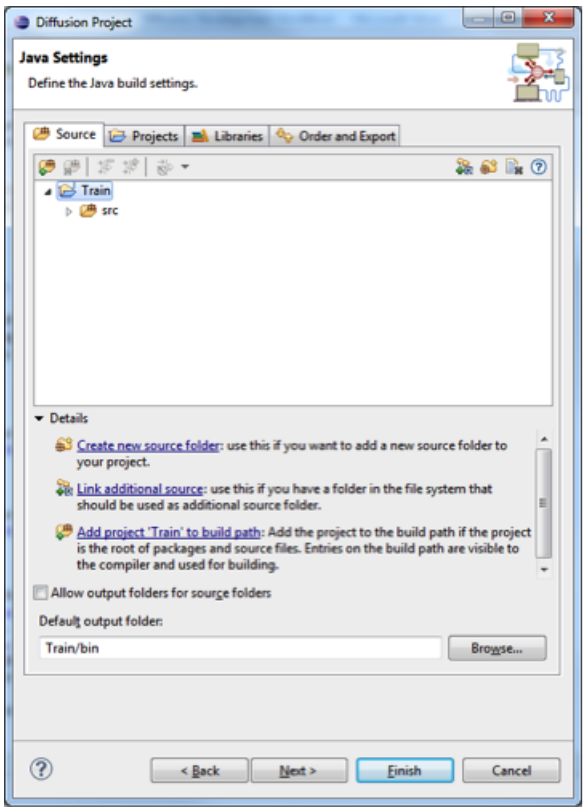


2. Highlight Diffusion project and select Next. The next screen available in the wizard will ask you to input a project name. Enter a project name, ensure you have selected the right Java JRE version available on your machine and select Next;

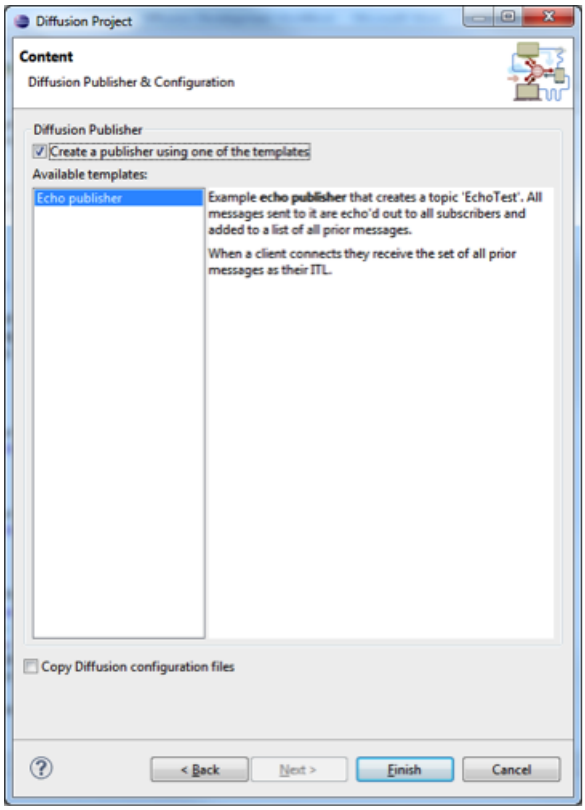


3. The next screen will ask you to configure any Java settings required for this project. You can skip this section and move forward, select Next;

Copyright 2013 Push Technology

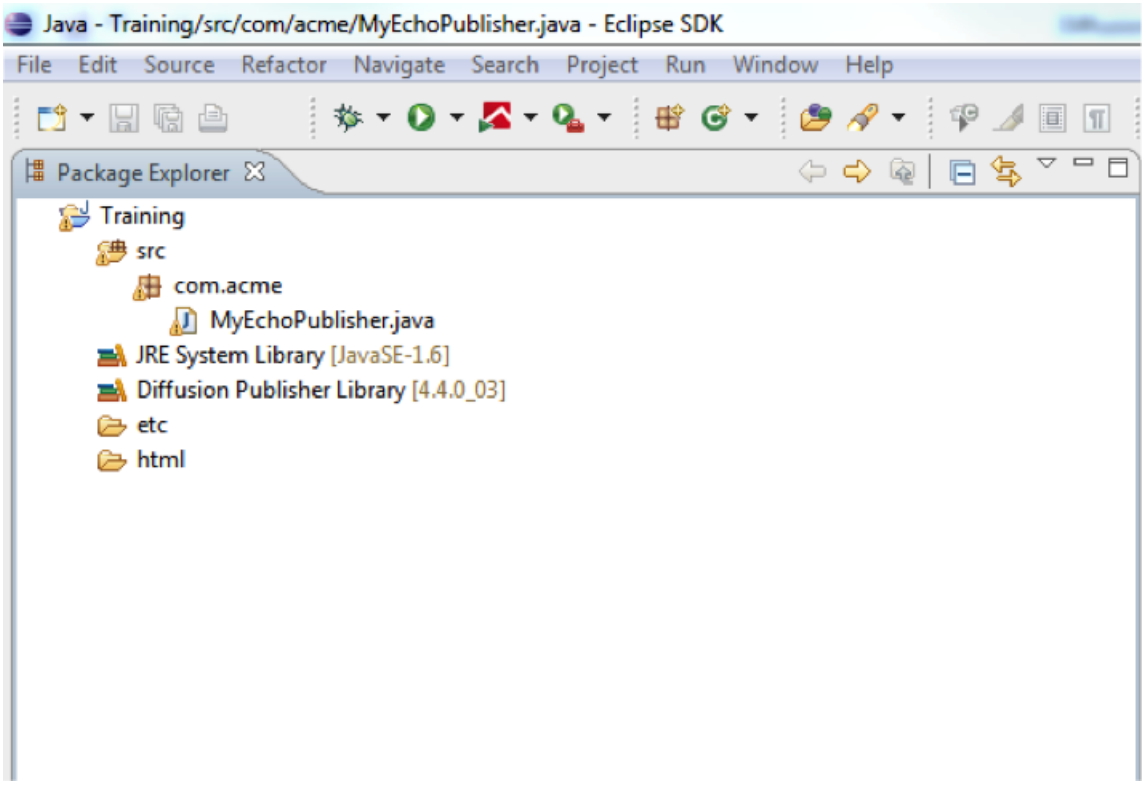

Diffusion Project
Java Settings
Define the Java build settings.
Source
Projects
Libraries
Order and Export
Train
src
Details
Create new source folder: use this if you want to add a new source folder to your project.
Link additional source: use this if you have a folder in the file system that should be used as additional source folder.
Add project 'Train' to build path: Add the project to the build path if the project is the root of packages and source files. Entries on the build path are visible to the compiler and used for building.
Allow output folders for source folders
Default output folder:
Train/bin
Browse...
< Back
Next >
Finish
Cancel

4. The final screen will ask if you want to use the simple publisher that is provided by Diffusion. Check the tick box ”Create a publisher using one of the Templates” and ensure the Echo publisher is highlighted, select Finish;


Diffusion Project
Content
Diffusion Publisher & Configuration
Diffusion Publisher
Create a publisher using one of the templates
Available templates:
Echo publisher
Example echo publisher that creates a topic 'EchoTest'. All messages sent to it are echo'd out to all subscribers and added to a list of all prior messages.
When a client connects they receive the set of all prior messages as their ITL.
Copy Diffusion configuration files
< Back
Next >
Finish
Cancel

5. You have created a Diffusion project and the example publisher that you included in your creation of the project will now be displayed in the explorer window within Eclipse.

Copyright 2013 Push Technology



Change the Publisher Code

For the purpose of this guide there are two types of data dependent applications that we are going to demonstrate. The design of your Diffusion application will determine how you publish data.

The first example is for eGaming and in-play betting. The data you will publish are betting odds for specific sporting events.

The second example is for FX trading and asset management. The data that you will publish for this vertical will be FX trading data. The reason for making this determination at the start of the development process is because once the decision has been made about the data structure you are going to implement, you will be able to build and construct the data model and the topic structure to facilitate your application.

The first thing you will need to do is design a data model that fits the requirements of your application. Once you have the data model you can then modify the publisher to reflect this. The samples that are provided in the next two sections are examples of what a typical topic structure would look like to facilitate the eGaming and FX Trading examples.

Design a Data Model

An In-Play Betting Example

It is predominantly the act of placing a bet on a specific event or market whereby a price is offered to the client that they can place a monetised bet against. The type of data you would typically publish for an in-play betting application would be pricing data, or pricing odds. With this in mind, an example of the topic structure that might be created to support this type of application would be:

```
CompanyName/SportsBook/Market/Event/Selection/DenPrice
CompanyName/SportsBook/Market/Event/Selection/NumPrice
CompanyName/SportsBook/Market/Event/Selection/Status
```

Copyright 2013 Push Technology

Examples:

```
PushTechnology/SportsBook/Football/Premiership/Liverpool vs Norwich/Home/
DenPrice
PushTechnology/SportsBook/Football/Premiership/Liverpool vs Norwich/Home/
NumPrice
PushTechnology/SportsBook/Football/Premiership/Liverpool vs Norwich/Home/
Status
```

```
PushTechnology/SportsBook/Tennis/USOPEN/Federer vs Nadal/FedererWin/DenPrice
PushTechnology/SportsBook/Tennis/USOPEN/Federer vs Nadal/FedererWin/NumPrice
PushTechnology/SportsBook/Tennis/USOPEN/Federer vs Nadal/FedererWin/Status
```

FX Trading example

This is typically providing a trade price to a currency pair. The type of data that is published for FX trading will include bid, offer, high, low etc. The topic structure for this type of data model would be completely different to the in-play betting topic structure. An example of an FX trading data model is:

```
CompanyName/Assets/AssetType/CurrencyPair/Bid
CompanyName/Assets/AssetType/CurrencyPair/Offer
CompanyName/Assets/AssetType/CurrencyPair/High
CompanyName/Assets/AssetType/CurrencyPair/Low
```

Examples

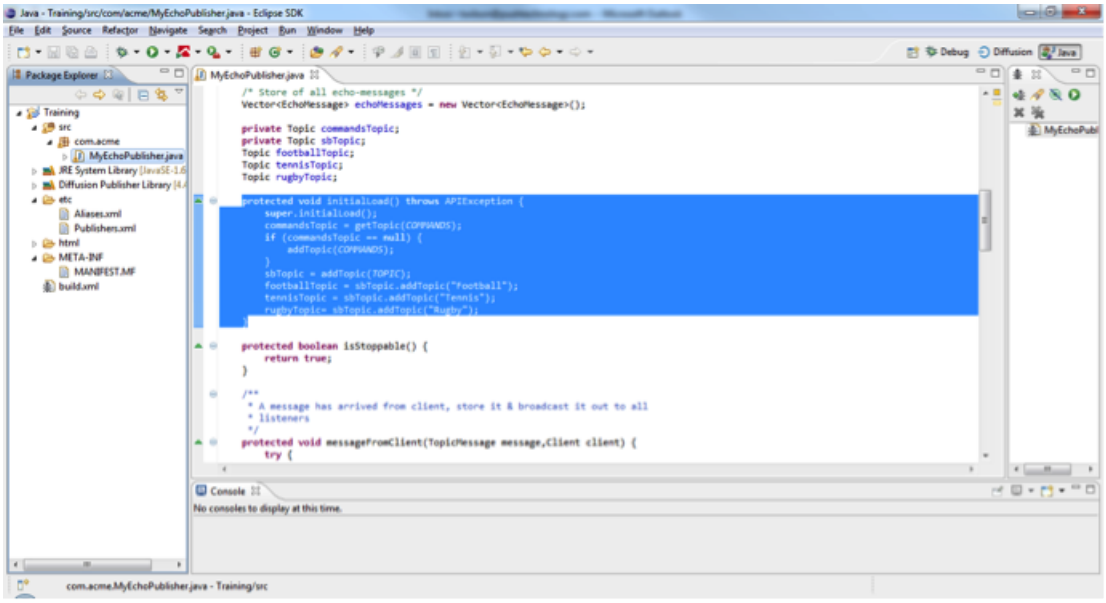
```
PushTechnology/Assets/FX/GBPUSD/Bid
PushTechnology/Assets/FX/GBPUSD/Offer
PushTechnology/Assets/FX/EURUSD/Bid
PushTechnology/Assets/FX/EURUSD/Offer
```

Once you have decided on the topic structure you want to build to support your application you are ready to start the development.

Modify the Call-Back Methods

Using your Eclipse IDE and the Diffusion project you created earlier, navigate to the src folder in the package explorer and open each branch until you locate the MyEchoPublisher.java file. Open this file and you will notice the source code displayed in another main window.

Copyright 2013 Push Technology



InitialLoad()

The first method you are going to modify is the initialLoad() method. This is the first method that is executed when a publisher starts and it is here that you are going to create the topic structure to support your data model.

If you have decided to build an in-play betting application you would typically create a sportsbook. Here is some sample code that you can use to build the topic structure for a sportsbook application and there is further guidance on how to modify the initialLoad() method to suit this application.

You will need to create a few object variables before you modify the call-back method. Create the class variables:

```
public static final String COMPANY_TOPIC = “PushTechnology”;
public static final String MAIN_TOPIC = "Sportsbook";
public static final String COMMANDS_TOPIC = "Commands";
Vector <EchoMessage> echoMessages = new Vector<EchoMessage>();

Topic companyTopic;
Topic commandsTopic;
Topic sportsbookTopic;
Topic footballTopic;
Topic tennisTopic;
Topic rugbyTopic;
```

The sample of code below will give you an idea about how to construct the topic structure.

```
protected void initialLoad() throws APIException {
     super.initialLoad();
     commandsTopic = getTopic(COMMANDS_TOPIC);
     if (commandsTopic == null) {
  addTopic(COMMANDS_TOPIC);
     }
     companyTopic = addTopic(COMPANY_TOPIC);
     sportsbookTopic = companyTopic.addTopic(MAIN_TOPIC);
     footballTopic = sportsbookTopic.addTopic("Football");
     tennisTopic = sportsbookTopic.addTopic("Tennis");
     rugbyTopic= sportsbookTopic.addTopic("Rugby");
```

Copyright 2013 Push Technology

```
}
```

The above sample of code will create the following topic structure;

```
PushTechnology/Sportsbook/Football
PushTechnology/Sportsbook/Tennis
PushTechnology/Sportsbook/Rugby
```

If you wanted to add the Premiership or USOpen as the next branch to the topic structure you would need to do the following;

Add additional class variables:

```
Topic premiershipTopic;
Topic usOpenTopic;
```

Add an additional `addTopic` method within `intialLoad()`:

```
premiershipTopic = footballTopic.addTopic("Premiership");
usOpenTopic = tennisTopic.addTopic("USOpen");
```

If the changes have been made to include the Premiership and USOpen as markets then the topic structure would be represented by:

```
PushTechnology/Sportsbook/Football/Premiership
PushTechnology/Sportsbook/Tennis/USOpen
```

If, on the other hand, you have decided to build an FX trading application then we have provided sample code and guidance on how to create an FX trading application and what needs to change in the initialLoad() method to reflect this data model.

You will need to create a few object variables before you modify the call-back method.

Create the class variables:

```
public static final String COMPANY_TOPIC = "PushTechnology";
public static final String MAIN_TOPIC = "Assets";
public static final String COMMANDS_TOPIC = "Commands";
Vector <EchoMessage> echoMessages = new Vector<EchoMessage>();

Topic companyTopic;
Topic commandsTopic;
Topic assetsTopic;
Topic FXTopic;
Topic GBPUSDTopic;
Topic EURUSDTopic;
```

The sample of code below will give you an idea about how to construct the topic structure.

```
protected void initialLoad() throws APIException {
 super.initialLoad();
 commandsTopic = getTopic(COMMANDS);
 if (commandsTopic == null) {
  addTopic(COMMANDS);
 }
 companyTopic = addTopic(COMPANY_TOPIC);
```

Copyright 2013 Push Technology

```
 assetsTopic = companyTopic.addTopic(MAIN_TOPIC);
 FXTopic = assetsTopic.addTopic("FX");

 GBPUSDTopic = assetsTopic.addTopic("GBPUSD");
 EURUSDTopic= assetsTopic.addTopic("EURUSD");
}
```

The above sample of code will create the following topic structure;

```
PushTechnology/Assets/FX/GBPUSD
PushTechnology/Assets/FX/EURUSD
```

If you wanted to add the Bid and Offer topics as children to the currency pair parent topics within the topic structure you would need to do the following;

Add an additional `addTopic` method within `initialLoad()`

```
GBPUSDTopic.addTopic("Bid");
GBPUSDTopic.addTopic("Offer");
EURUSDTopic.addTopic("Bid");
EURUSDTopic.addTopic("Offer");
```

If the changes have been made to include the Bid and Offer as prices then the topic structure would be represented by:

```
PushTechnology/Assets/FX/GBPUSD/Bid
PushTechnology/Assets/FX/GBPUSD/Offer
PushTechnology/Assets/FX/EURUSD/Bid
PushTechnology/Assets/FX/EURUSD/Offer
```

You can see how easy it is to create an entire topic structure to represent the data model you have developed. Try designing your own topic structure either by creating a whole new structure or using the examples above and enhancing them to suit your application. The main aim for the initialLoad() method is to have a topic structure that will support your data model and the data you are going to be publishing to the connected clients.

MessageFromClienty()

On completion of the changes to intialLoad() we will move onto another method that needs to be modified to suit the needs of your application.

For the purpose of this example you are going to change this method so that it will allow messages to be received from any connected clients, received into the Diffusion server and then published to any client that has subscribed to the topic that the messages relates to.

The sample code below shows how Diffusion receives a message from a client, populates a vector of type EchoMessage, associates the message with the relevant topic and then publishes the message to all connected clients who have previously subscribed to that topic.

```
protected void messageFromClient(TopicMessage message,Client client) {
 try {
  EchoMessage echoMessage = new EchoMessage(message,client);
  echoMessages.add(echoMessage);
  Topic topic = getTopic(message.getTopicName());
  topic.publishMessage(message);

 }
 catch (Exception ex) {
 this.logWarning("Exception caught",ex);
```

Copyright 2013 Push Technology

```
  }
}
```

Change the code in this method to reflect the sample above and your application will have the ability to receive message from any connected clients, for a specific topic, and publish that message to any connected clients, who have subscribed to that topic (shown an interest).

MessageFromEventPublisher()

The final method to allow data to be received from a data source, through Diffusion, and out to all connected clients is the messageFromEventPublisher() method. This method is predominantly used to receive topic messages from an external data source, an event publisher, process the message by associating the message to a topic that exists within the publisher and finally publishing the message on to any clients that have subscribed to the topics.

The sample code below demonstrates how Diffusion accepts the message, associates it with a topic and then publishes the message to the clients.

```
@Override
protected void messageFromEventPublisher(EventConnection eventConnection,
 TopicMessage topicMessage) {
    try {
        String topicName = topicMessage.getTopicName();
        Topic topic;
     topic = getTopic(topicName);
        if(topic == null) {
       SingleValueTopicData topicData =
       TopicDataFactory.newSingleValueData(MDataType.STRING);
            topic = addTopic(topicName, topicData);
     }
        topic.publishMessage(topicMessage);
    }
    catch (Exception e) {
        this.logWarning("Exception caught",ex);
    }
}
```

Modify this method to reflect the sample above and your application will have the ability to receive message from an external event publisher, for a specific topic, and publish that message to any connected clients, who have subscribed to that topic (shown an interest).

Building a Deployment File

Once you have completed the development of your publisher you will be able to package the source code into a .DAR file, a Diffusion Archive file that can be deployed within any instance of Diffusion. Whether you are using a cloud offering or you are going to run the Diffusion application on a server the deployment process is exactly the same.

The first thing you need to do is build a .DAR file and the sample code below demonstrates how to achieve this using an ANT build script, which can be executed within your Eclipse development environment. This is an example of what the script needs to execute:

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="MyEchoPublisher" default="deploy">
    <property name="publisher.name" value="MyEchoPublisher" />
    <property name="jar.name" value="${publisher.name}.jar" />
    <property name="diffusion.dir" value="C:\Diffusion\Diffusion4.4.0_03" />
    <property name="dar.name" value="${publisher.name}.dar" />

    <target name="makejar">
        <jar jarfile="${jar.name}" includes="**/*.class" basedir="bin" />
```

Copyright 2013 Push Technology

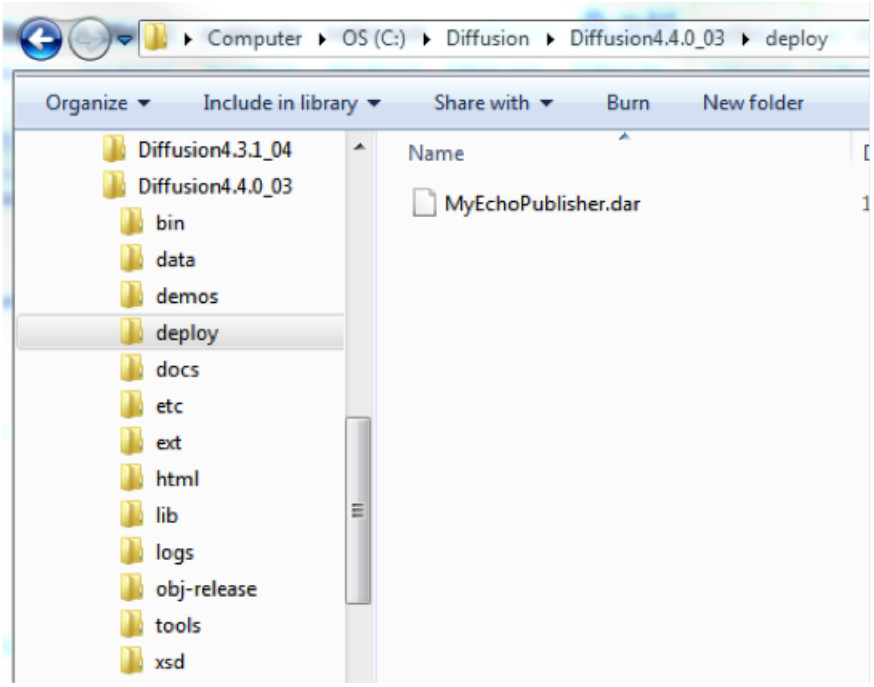
```
    </target>

    <target name="makedeployable" depends="makejar">
        <delete dir="deploy" />
        <mkdir dir="deploy/${publisher.name}" />
        <copy todir="deploy/${publisher.name}">
            <fileset dir=".">
                <include name="data/**" />
                <include name="etc/**" />
                <include name="ext/**" />
                <include name="html/**" />
            </fileset>
        </copy>
        <copy todir="deploy/${publisher.name}/ext" file="${jar.name}" />
        <jar jarfile="${dar.name}" includes="${publisher.name}/**"
 basedir="deploy" manifest="META-INF/MANIFEST.MF" />
    </target>

    <target name="deploy" depends="makedeployable">
        <mkdir dir="${diffusion.dir}/deploy" />
        <copy todir="${diffusion.dir}/deploy" file="${dar.name}" />
    </target>
</project>
```

The above example will result in the creation of a .DAR file with the same name as the publisher class.

The .DAR file is copied into the deploy folder that is in the “diffusion.dir” property mentioned in the build script. This is the file you will need to deploy onto the server / cloud that is running your Diffusion instance.



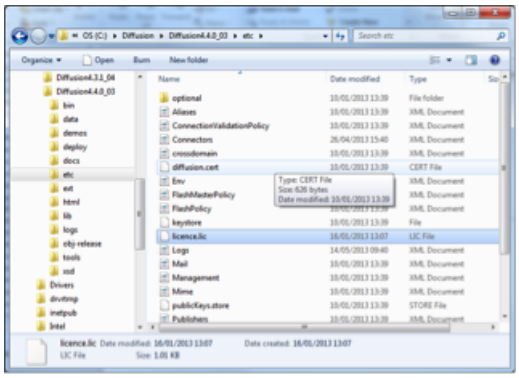
Deploying a Publisher

Once you have created the publisher .DAR file you can then deploy the file into a Diffusion instance, and Diffusion will start the publisher after checking that the code has compiled and initial software checks have been carried out. There are two things that need to be completed during the deployment phase.

Copyright 2013 Push Technology

Licence File

The first is to copy the licence.lic file from your local machine (found in the etc directory) to the etc directory of the Diffusion instance you want to start on the server you want Diffusion to run on. This is regardless of whether Diffusion is running on an actual server or within the cloud. You must ensure that the file is copied into this folder before you start Diffusion. Diffusion will not start without a valid licence.



publisher File (.DAR)

The second is to copy the .DAR file from your local machine to the deploy directory of the Diffusion instance you are going to start on the server you will be running Diffusion on. This .DAR file is located in the deploy directory of your local installation of Diffusion, only if you used the ANT script previously discussed. This is regardless of whether Diffusion is running on an actual server or within the cloud.

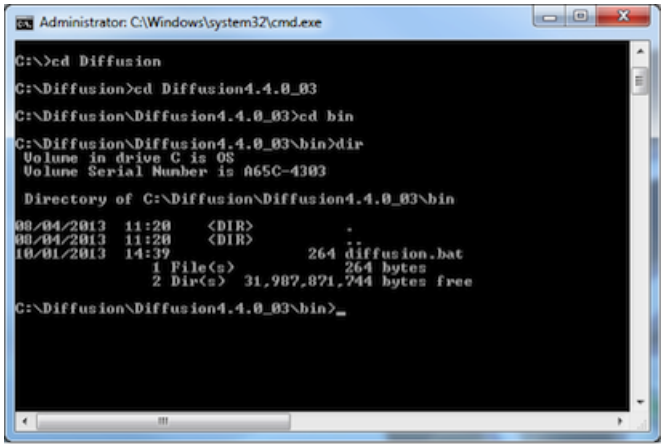
Starting Diffusion

Regardless of whether you are running Diffusion in the cloud or on your own server the method used to start Diffusion remains the same.

Note: If you are running Diffusion in the cloud you will need to be able to SSH into the machine and permit yourself root access in order to execute a start script to initiate Diffusion.

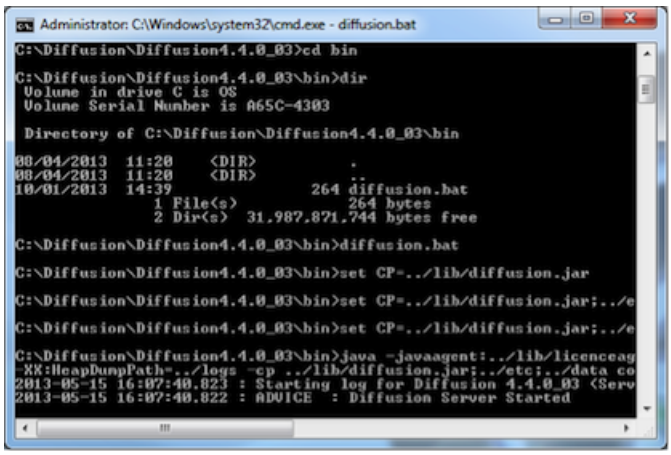
A Diffusion start script located in the bin directory of the installed Diffusion software can be executed via the command line. The steps below demonstrate how to execute this script;

1. Open a command prompt and navigate to the bin directory of the installed Diffusion software. The file is called diffusion.bat.



2. Run the diffusion.bat to start Diffusion.

Copyright 2013 Push Technology



The confirmation message you now want to see is **"Diffusion Server Started"**

Generating Data Using the Event Publisher Tool

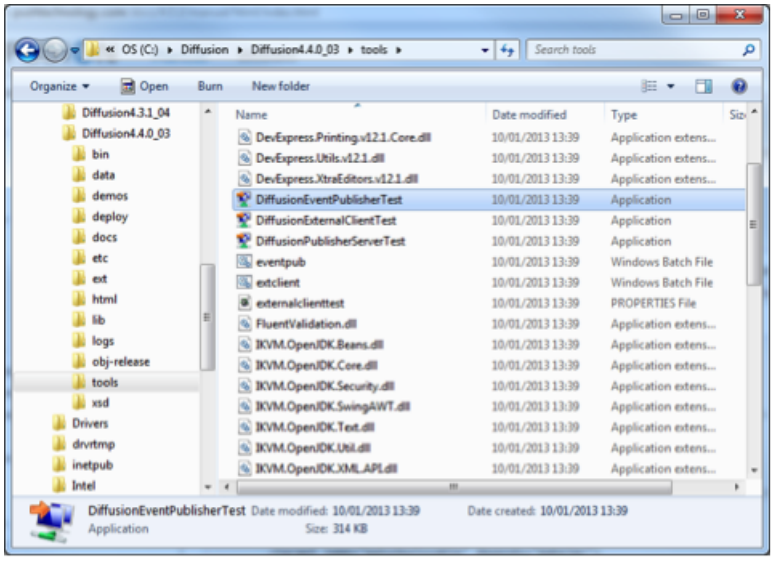
Now that you have deployed your publisher, the next step is to generate sample data that can be sent to any of the topics that are operating within the publisher that you have created. Push Technology provide a number of tools that can be used as part of the testing process to allow you to analyse and understand how your publisher is going to perform.

One of the tools provided is an external event publisher. By using this tool you will be able to send data to the publisher as though it was a data source providing data to your Diffusion application.

The event publisher accepts data from a data source, transforms the data into Diffusion topics and topic data and then sends a topic message to your publisher. The publisher will accept the message, associate to a topic and then publish the message, including the payload (actual data), to any clients that have subscribed to that topic.

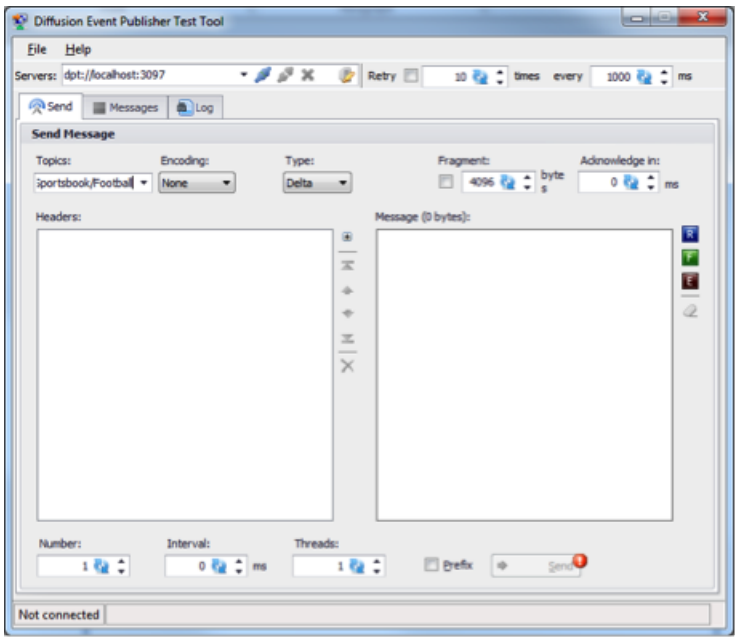
Follow the steps below to understand how to use the external event publisher tool and generate data to publish to your deployed publisher.

1. Open a file explorer window and navigate to the installed Diffusion product file directories. Go to the tools directory and locate the DiffusionEventPublisherTest application.

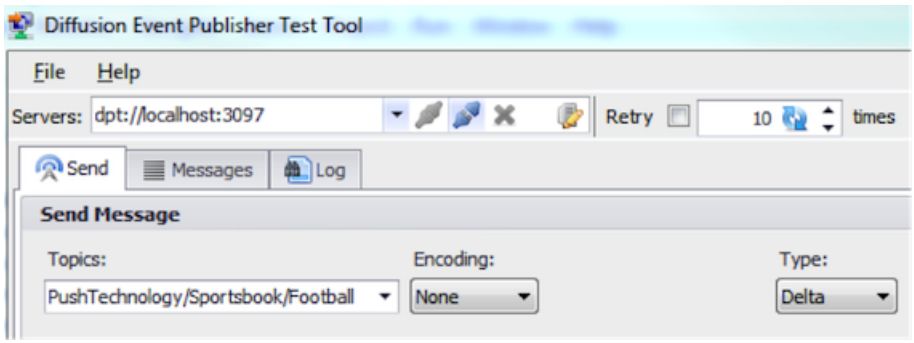


2. Start the external event publisher tool by either double clicking on the file or selecting the file, right clicking and choosing Run as Administrator option.

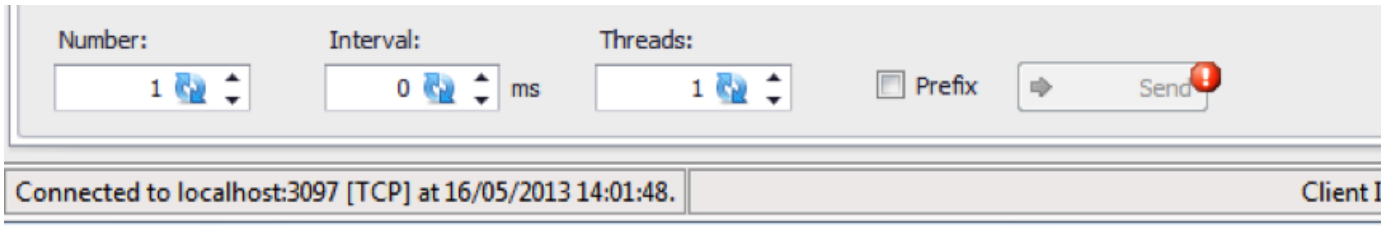
Copyright 2013 Push Technology



3. Enter a connection URL in the box provided for connection to a server. This needs to represent the server that Diffusion is running on. Typically you will enter an IP address and the port number needs to be 3097.



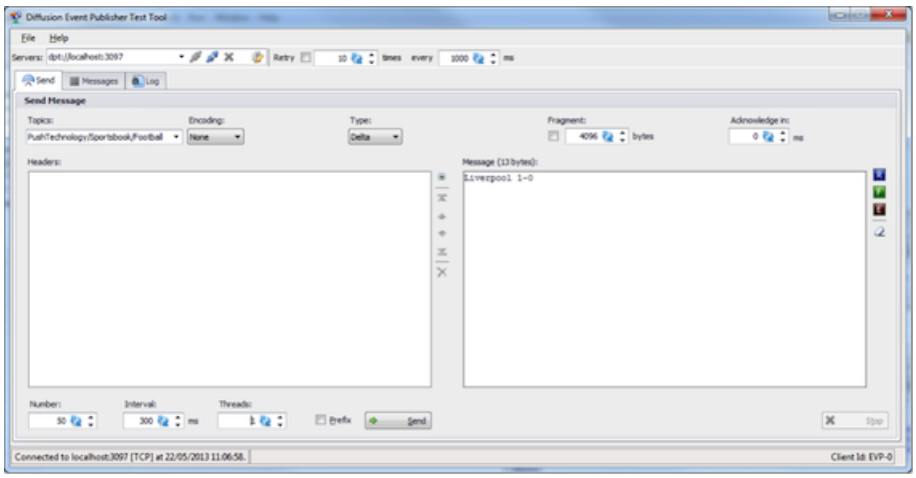
4. Once a connection has been established the connection information is display in the bottom panel of the tool window.



5. Now that the connection has been established, you can enter a specific topic that you want to generate data for and this must be a topic that exists in the publisher you created and deployed previously. Once you have entered the topic name, in the message window, type in the message you want to send and then finally enter how many messages you want to send and at what interval. This is done at the foot of the main window.

   The example below is going to generate 50 messages every 300 milliseconds with a prefix on the message so you can identify the message number when it is received on the client. The topic that will have messages generated to it will be Sportsbook/Football and the message(s) will be Liverpool 1-0.

Copyright 2013 Push Technology



The event publisher will send these topic messages to the Diffusion publisher you have created, accept the topic and its associated data and publish the message on to all the clients that have subscribed to that topic. The next section will provide guidance on how you can view that data from a client's perspective.

Viewing the Data

Now that you have built, deployed and started your Diffusion application you are now in a position to subscribe to the data that Diffusion is publishing. There are two ways available to viewing the data, however in both approaches the concept of a client is exactly the same and they both provide the same features and functionality.

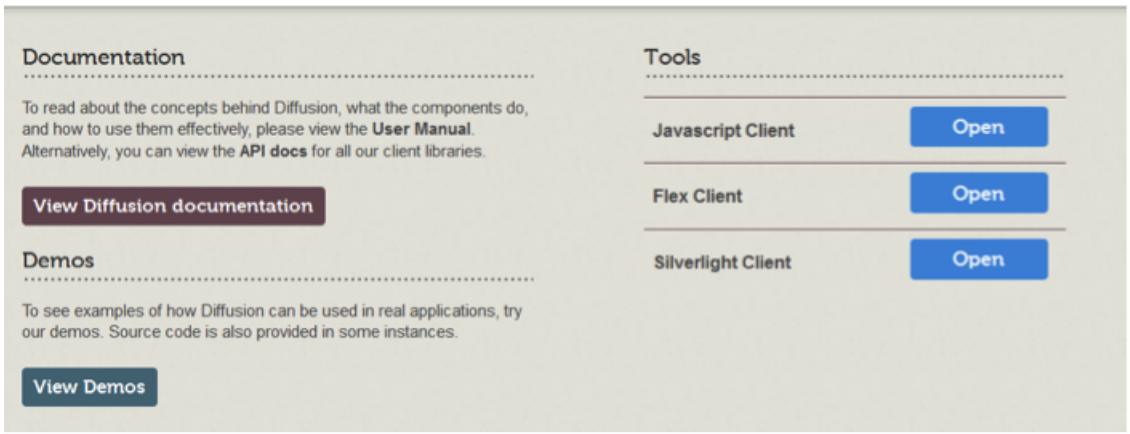
Using an External Client Tool

The final step to your Diffusion application is to open a client front end and subscribe to the topics that you have created in your publisher. Once you have subscribed to the topics you will be able to see the messages being published to the client.

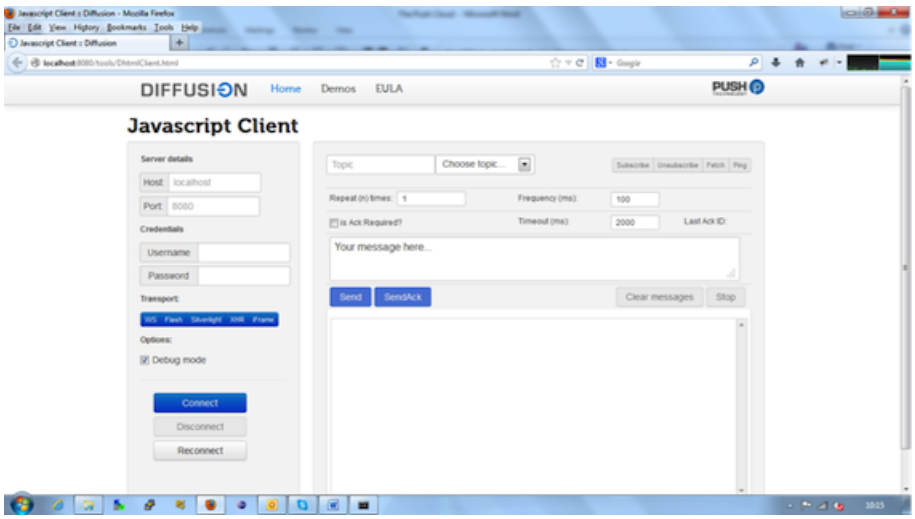
This section will guide you through a client front end initiation and shows you how to subscribe to the topics that you want to receive messages for.

By way of subscription you are telling Diffusion what information / data you are interested in and Diffusion will only publish updates to that specific piece of information / data as and when it changes. These messages are known as 'deltas'.

1. By way of subscription you are telling Diffusion what information / data you are interested in and Diffusion will only publish updates to that specific piece of information / data as and when it changes. These messages are known as 'deltas'.

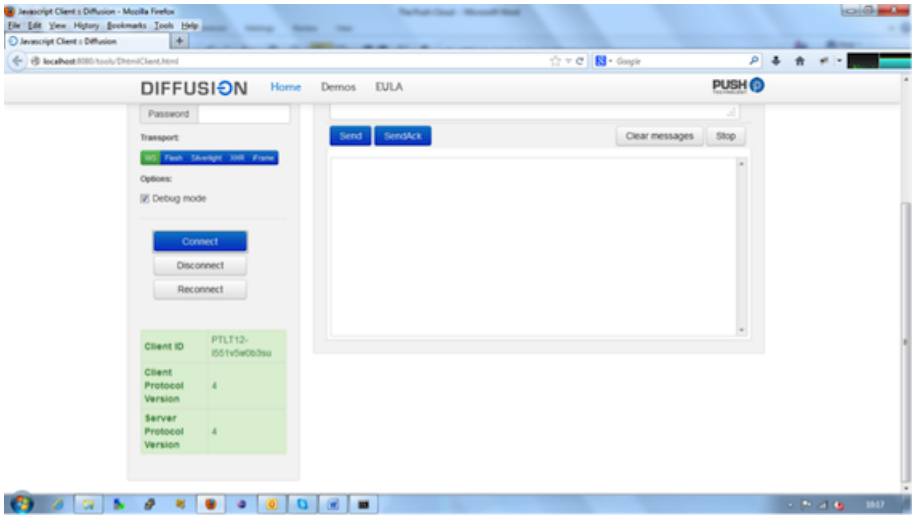

Documentation
To read about the concepts behind Diffusion, what the components do, and how to use them effectively, please view the User Manual. Alternatively, you can view the API docs for all our client libraries.
View Diffusion documentation
Demos
To see examples of how Diffusion can be used in real applications, try our demos. Source code is also provided in some instances.
View Demos
Tools
Javascript Client
Open
Flex Client
Open
Silverlight Client
Open

Copyright 2013 Push Technology



2. In the host field, enter the IP address and the port number of the server that Diffusion is running on. If it is running on a physical server then enter the IP address of that server.

   If it is running in the cloud then enter the elastic IP address associated to the Diffusion instance running in the cloud. Select the WebSockets connection from the transport options available to you and then click Connect. Once the connection has been established you will notice the client information displayed in a panel on the left hand side (in green).



3. Enter the name of the topic that you want to subscribe to. This will be the same name that you are publishing messages to from the publisher that you created previously.

4. d) You will see in the messages box as soon as you have successfully subscribed to a topic the messages that are published from the event publisher you have started via the Diffusion publisher that you deployed will start appearing in the main window.

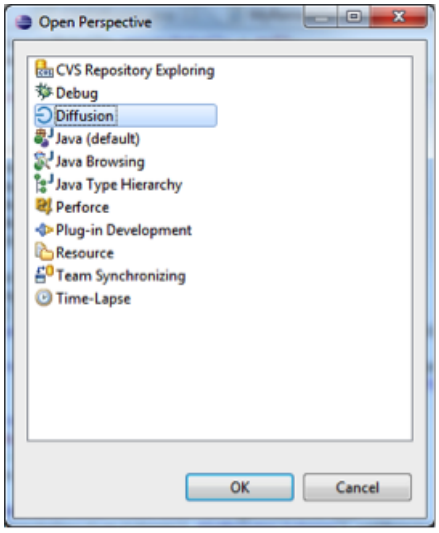
Copyright 2013 Push Technology

You will notice the messages that are displayed in the message window reflect the data that is being published from a data source to the Diffusion publisher running in the cloud and then to the front end client.

Using Introspector

Introspector is a development tool that allows you to deep dive into the Diffusion instance you have deployed and started. It provides windows (views) that you can use to connect to the publisher and then subscribe to the topics and watch the data that is delivered by Diffusion. Using this tool you will be able to identify what data is being published, the frequency of the updates and also what clients have connected to the server and which topics they are interested in.

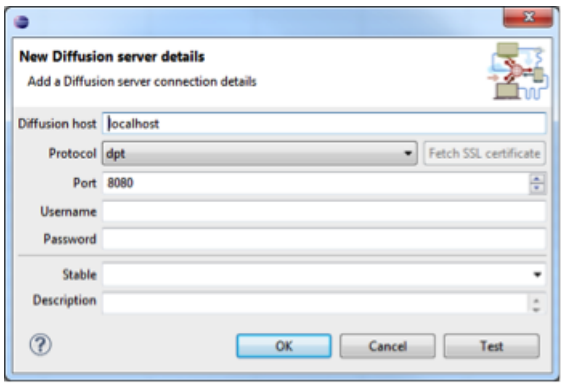
To use Introspector within Eclipse you will need to open the Diffusion Perspective. From the window menu select the “Open Perspective”, then Other and select the Diffusion item. Opening the Diffusion perspective will present you with a set of views related to Diffusion.



Once you have the Diffusion perspective open you can then create a connection to the Diffusion instance that you want to view.

To create a connection, in either the client tab or the topic tab, select the green “PLUS” sign and enter the details of the server you want to connect to. You can either enter localhost if you want to connect to an instance of Diffusion running locally or you can specify the IP address of a service (perhaps the cloud) that is running Diffusion.

Copyright 2013 Push Technology



Click on the test button to ensure a connection can be established.

There are two views available within Introspector:

- The Clients View
- The Topic View

The topic view gives the users the ability to view the Diffusion topic structure for every publisher that is running on the server they have created a connection to.

Users can navigate the tree of topics as they would any other tree control.

The GUI populates the tree on-demand and asynchronously, so it's possible that a branch of the tree may populate some short time after it has been opened.

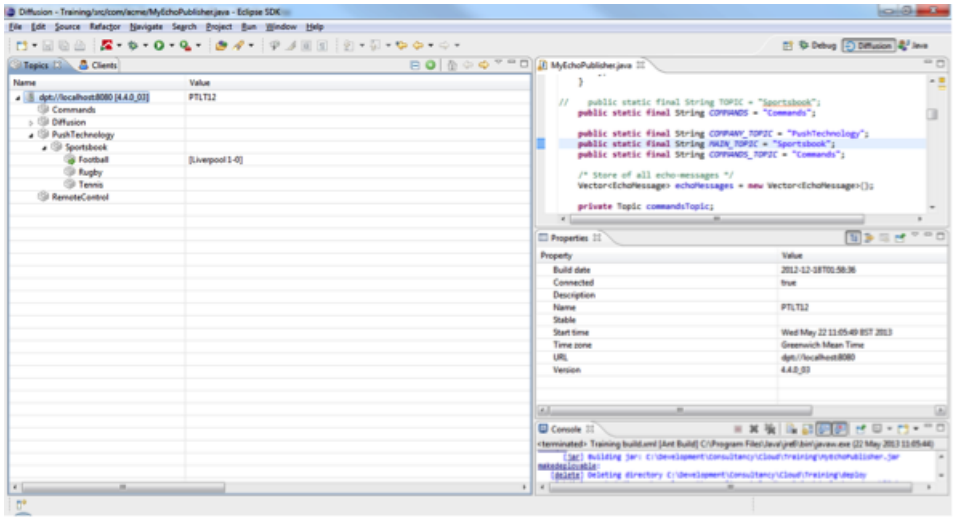
Each topic shows its name, the publisher, the type of any topic-data associated, the date-stamp the topic was created, the date-stamp the topic was last updated and the value of that topic. The last two will only be populated if the topic has been fetched or subscribed to.

The topic tree is live and as new topics are created and old topics deleted so the tree will be updated.

Users may retrieve the current state of the topic by picking 'Fetch' from the context menu of a topic. Picking 'Fetch recursively' will also fetch the values of all topics below this one.

Users may also subscribe to a topic by picking 'Subscribe' from the context-menu of a topic. 'Subscribe recursively' will also subscribe to values of all topics below this one.

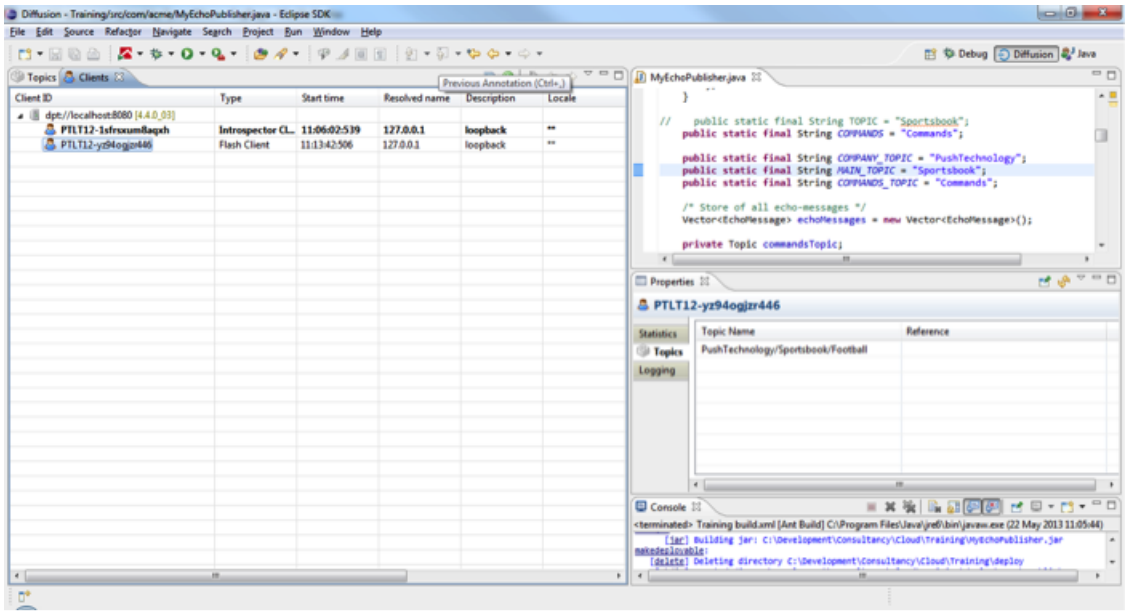
If the value of a topic changes, the Value and Updated columns flash blue for a second. If the user wishes to drill into a part of the topic tree they may pick 'Go Into' from the topic context menu. The view then shows the name of the root topic-name in the view header.



The clients tab allows you to view information about all of the current connected clients to Diffusion.

Copyright 2013 Push Technology

This view allows you to analyse each connected client including how many messages the client has sent and received, how many topics they have subscribed to and also the type of client connection.



There are three menu options available per client:

- Properties - Provides a view to messages sent, received, and topics subscribed too.
- Monitor - Allows useful information about Diffusion and the underlying OS, this includes memory utilisation, memory allocation and OS version.
- Configuration - Allows you to view the configuration for the Diffusion instance running on the server you have created a connection too.

Appendix - Further Information

Diffusion Overview

Diffusion Documentation

Download Diffusion

Download Eclipse IDE

Copyright 2013 Push Technology

Chapter 4

Overview

Topics:

- *Architecture*
- *Diffusion™ Server*
- *Messaging*
- *Clients*
- *Publishers*
- *Topics*
- *Message Queues*
- *Concurrency*
- *Glossary*

An Overview of the Diffusion™ product

This section provides an overview of the Diffusion™ product for those attempting to gain an initial understanding of the various components and functionality. The following concepts are introduced:-

- Architecture
- The Diffusion™ Server
- Messaging
- Clients
- Publishers
- Topics
- Message Queues
- Concurrency

Finally, a Glossary is provided.

Architecture

An architectural overview of Diffusion™

Basic Architectural Components

Diffusion™ is primarily a Distributed Message Broker infrastructure. Some of the main components are introduced in the diagram below:-

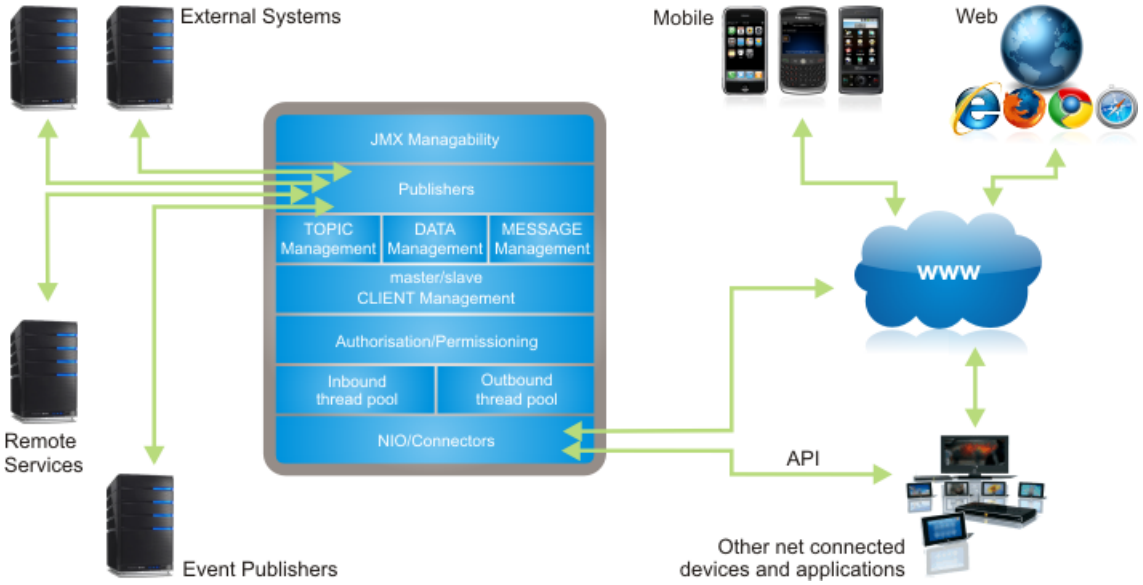


Figure 10: Basic Components

Many of the components shown are optional. Only Publishers and Clients are essential for a working Diffusion™ application.

Clients connect to a Diffusion™ Server and Publishers publish Messages to Clients that are interested in those Messages. The communication is bi-directional and thus Clients may also send Messages to Publishers. A Publisher is a user written Java class.

An Event Publisher allows for data to be fed to Publishers from some external application.

A Remote Service can perform some of the functions of a Publisher remotely. For example, creating Topics, authorising Clients and their subscriptions, subscribing Clients and of course publishing Messages.

In a distributed environment a Publisher can connect to a Diffusion™ Server as a Client.

Publishers and Clients are loosely coupled - the link being provided by Topics

The Diffusion™ Server also provides Web Server capabilities via its Connectors.

Distribution

Diffusion™ Servers may communicate in a hub and spoke architecture to provide full scalable distribution . Load may be spread across different Publishers in different Servers across different machines. Publishers may act as Clients of other Servers by subscribing to Topics in the same way as normal Clients.

The following diagram shows a typical distributed configuration:-

Copyright 2013 Push Technology

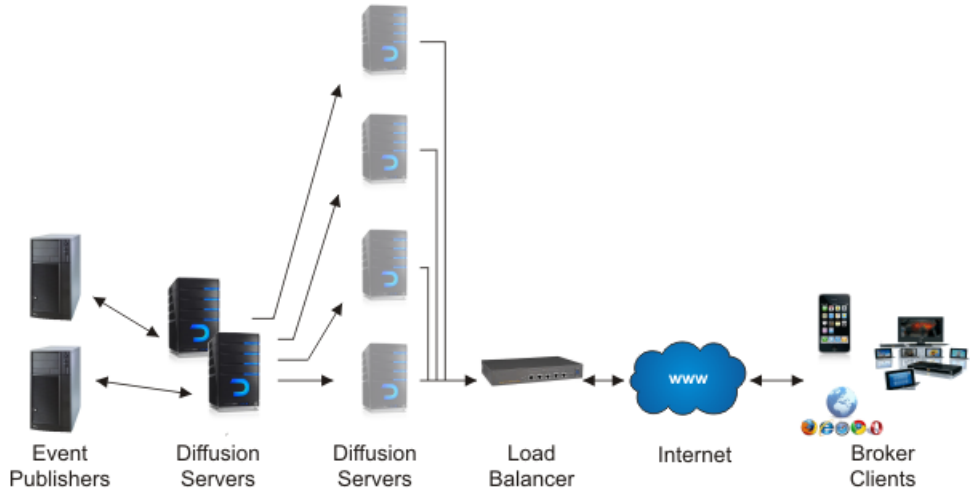


Figure 11: Distribution

Diffusion™ Server

An overview of the Diffusion™ Server

The Diffusion™ Server is a highly efficient, low latency messaging server which provides the following main functions:

- A scalable and mature solution to push (stream) and receive data and events, in real-time, both to/from a web browser and other net connected devices.
- High performance transport of data between two or more machines within your own network or extranet.

A single Server may be deployed to push messages to web browsers (or other net devices) or many interconnected Servers can provide a fully scalable enterprise messaging solution.

The Diffusion™ Server has been certified to run on the Oracle Java 7 Developer Kit (JDK/JRE).

The following diagram shows some of the key components of a Diffusion™ Server.

Copyright 2013 Push Technology

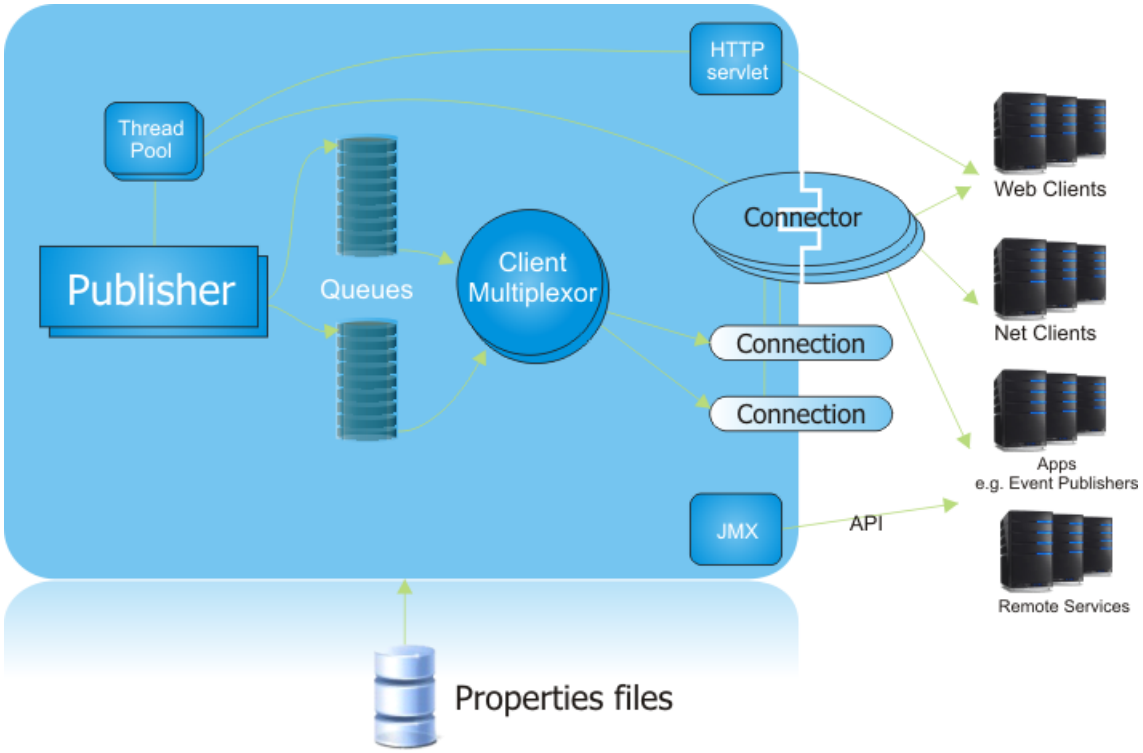


Figure 12: Server

The main purpose of the Server is to provide the environment in which one or more Publishers can run. Publishers provide Topics through which they publish Messages . Clients subscribe to Topics so that they may receive Messages from those Topics.

Clients connect to Diffusion™ via 'Connectors'. There can be any number of Connectors each listening on a different port. This provides a high level of control over connections. A Connector can act as a gateway allowing many different types of Client (HTTP, Browser, Plugin, API etc) to connect to a single connection point listening on any port.

Connectors also provide connection points for Event Publishers and for serving policy file requests.

In addition the Diffusion™ Server has Web Server capability.

Thread Pools are used for processing incoming Messages so that a thread is not used for each connection. Thread Pools may also be used by Publishers for background tasks.

Publishers publish (or send) messages onto Client queues where Client Multiplexer threads pick them up and send them to the Clients.

A JMX System Management interface is provided.

The functionality of a Server is configured via a set of XML properties files or alternatively the server may be instantiated in a Java application and configured programatically.

Messaging

Diffusion™ is essentially a Message Broker and as such provides a high performance infrastructure for sending Messages.

In Diffusion™ terms a Message can be defined as a contiguous sequence of bytes comprising some header information followed by a data payload.

There is no limit on the format of data held within a message.

Copyright 2013 Push Technology

A Message will contain data for only one Topic .

Messages are sent between Server, Clients and other components over socket based connections using the Diffusion™ Message Protocol.

Messages may be encoded to either compress or encrypt the data content of the Message.

The Java API provides a comprehensive set of functions for use by Publishers (and other Java API users) for creating, populating and reading Messages. Other APIs provide similar facilities.

Messaging Possibilities

Publisher to Client

The following messaging possibilities are provided from Publishers to Clients:-

- Publisher publishes Messages to all Clients subscribed to a Topic.
- Publisher publishes Messages to all Clients subscribed to a Topic except a specified Client.
- Publisher sends Message to individual Client.
- Publisher sends Message to a defined group of Clients.
- Publisher sends Message containing current state of Topic to Client in response to a 'fetch' request (see below).

Any of the above may be sent 'requiring acknowledgement' or at other than normal priority.

Client to Publisher

A Client may send a Message on a Topic which the Server will route back to the Publisher of that Topic. Such Messages mey be set as ' requiring acknowledgement'.

A Client may send a request for the current state of a Topic (fetch) which will be routed to the Publisher of that Topic which returns a state Message to the Client. A Client may do this even if not susbcribed to the Topic or even if the Topic does not exist (assuming Publisher caters for this).

Event Publisher to Publisher

An Event Publisher may send Topic Messages to a Server which will be routed to the Publisher of that Topic. A Publisher may also send Messages to an Event Publisher.

Remote Service to Server to Client

Remote Services may publish Messages to Topics or send Messages to individual Clients in the same way as a Publisher can. The Topics themselves are actually maintained at the Server.

A Remote Service may also send Messages to the Publisher that supports it in order to perform some action at the Server.

Client to Server to Renmote Service

If a Client sends a Message on a Topic that is managed by a Remote Service then the Message will be routed to the Remote Service.

Clients

At the core of Diffusion™ is a Server which is connected to by Clients. A Client can be defined as any application communicating with a Diffusion™ Server using a Diffusion™ Client Protocol.

A Diffusion™ Server hosts Publishers that push (or publish) Messages to Clients. A Publisher may send Messages directly to a Client (or group of Clients) or may publish Messages to all Clients that are subscribed to a particular Topic. Diffusion™ provides client APIs for a number of different platforms which enable applications in many different environments to act as Diffusion™ Clients. A Diffusion™ Server can support a very large number of concurrently connected Clients.

Copyright 2013 Push Technology

A Diffusion™ Server hosts Publishers that push (or publish) Messages to Clients. A Publisher may send Messages directly to a Client (or group of Clients) or may publish Messages to all Clients that are subscribed to a particular Topic .

Types of Client

Some of the types of Client that are directly supported by Diffusion™ are mentioned below.

Web Clients

Clients can be browser applications utilising one of the following interfaces provided by Diffusion™:-

- HTTP Callback or IFrame
- Flash
- Silverlight
- Websocket

Net Clients

A Client can be any application connecting to Diffusion™ over a DPT socket connection which can be over the internet or an intranet/extranet. Java and .NET and external client APIs are available. Applications in other languages could communicate using the raw DPT.

Mobile Clients

APIs are available for the following mobile devices:-

- Blackberry (J2ME)
- iOS
- Android
- Windows phone.

Publisher Clients

A Publisher can connect to another Diffusion™ Server and subscribe to Topics there, causing that Publisher to become a Client. This type of communication provides for a distributed Diffusion™ system.

Remote Services

A Remote Service is a specialised form of Client that is able to perform some of the functions a Publisher could at the server. For example it is able to create Topics, authorise Client connections, authorise Client subscriptions and subscribe Clients to Topics etc.

Publishers

Publishers are components hosted within a Diffusion™ Server which manage the data for one or more Topics and publish Messages transmitting a snapshot of the current state of that data to any Client that subscribes to the Topic and then any subsequent changes to that state.

Publishers are written using the Java API and must extend the issued `Publisher` class and implement various methods to provide the Publisher functionality. Only as much as is required need be implemented and writing a Publisher can be a very simple task.

A Publisher needs to maintain its own data model. The Publisher may initialise its data as it starts and then update it as a result of external events. The updates to the data may be received by the Publisher in any way that is appropriate but Diffusion™ provides the Event Publisher facility which can feed messages directly into Publishers as well as the Remote Service facility which as well a feeding Messages can manage Topics and Clients.

When a Client first subscribes to a Topic the Publisher can provide the Client with a snapshot of the current state of the data relating to that Topic. This is called a ' Topic Load '.

Copyright 2013 Push Technology

A Client may also request the current state of a Topic, even if not subscribed to it, using the 'fetch' command. There is even the possibility of a Publisher responding to a fetch request of a Topic that does not exist thus providing a potential request/response mechanism without the overhead of real Topics.

A Publisher will maintain any changes to its Topic data state and publish those changes to the Topic as 'delta' Messages. This results in the Message being sent to every Client that is subscribed to the Topic.

There is also the option to publish a Message to all but a specified Client which can be useful when the publishing of a Message is in response to a Message from that Client.

Publishers may also send Messages to individual Clients or to groups of Clients.

For normal Publishing the Publisher does not need to know or keep track of the Clients subscribed to its Topics.

Publishers may also receive Messages from Clients.

Publishers own the topics they create. Ownership of a topic is used to determine which Publisher receives a Message from a Client, deals with subscription and when creating hierarchical topics.

Topics

Clients and Publishers are loosely coupled via logical links called Topics.

A Publisher publishes Messages to a Topic and a Client subscribes to a Topic and receives its Messages. Each Topic is owned by a Publisher. A Client can also send Messages to a Topic which routes them back to the owning Publisher.

The Client is not aware of the Publisher, only the Topic. The diagram below shows the relationship between Publishers and Clients via Topics:-

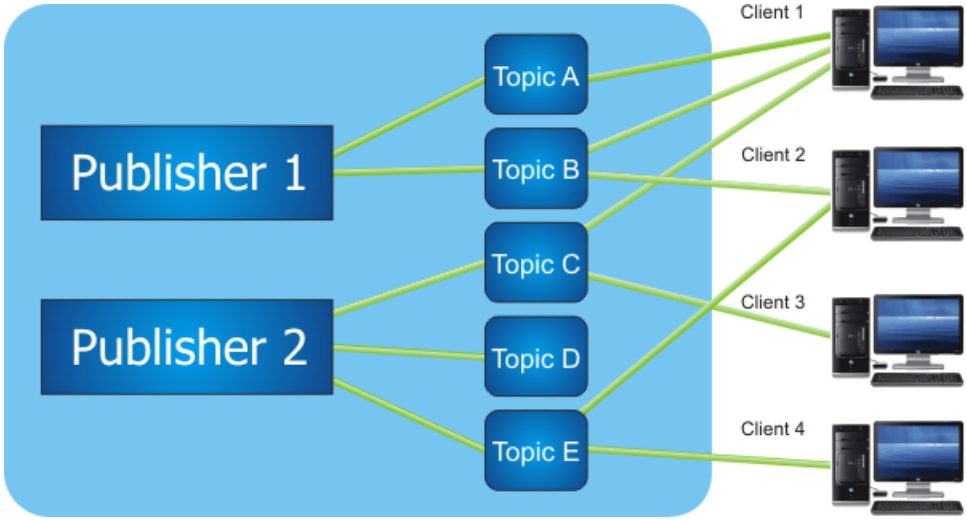


Figure 13: Topics

Topics are created in a Diffusion™ Server by Publishers. Each Topic must have a unique name within the Server.

Topics may be arranged hierarchically with each Server maintaining a tree of Topics.

A Publisher may provide any number of Topics and a Client may subscribe to any number of Topics.

Message Queues

Every Client has it's own Message Queue which queues references to the Messages to be sent

Publishers Publish Messages to Topics and all Clients subscribed to those Topics will receive the Messages. Publishers may also send Messages to individual Clients or logical groups of Clients. Clients may send Messages to Topics which are routed back to the Topic's Publisher.

Copyright 2013 Push Technology

Every Client has its own Message Queue within the Server and when a Publisher publishes a Message it is put on the Queue of each Client subscribed to the Message Topic. The same applies to individual Messages to Clients. A Client's queue could therefore have Messages from more than one Publisher on it and they will be sent to the Client strictly in the order they were enqueued.

Diffusion™ does not copy the Message data for each Client that the Message is to be sent to. Only pointers to the data are put on queues and the data remains until there are no more pointers to it.

The following diagram shows how Messages are queued for Clients:-

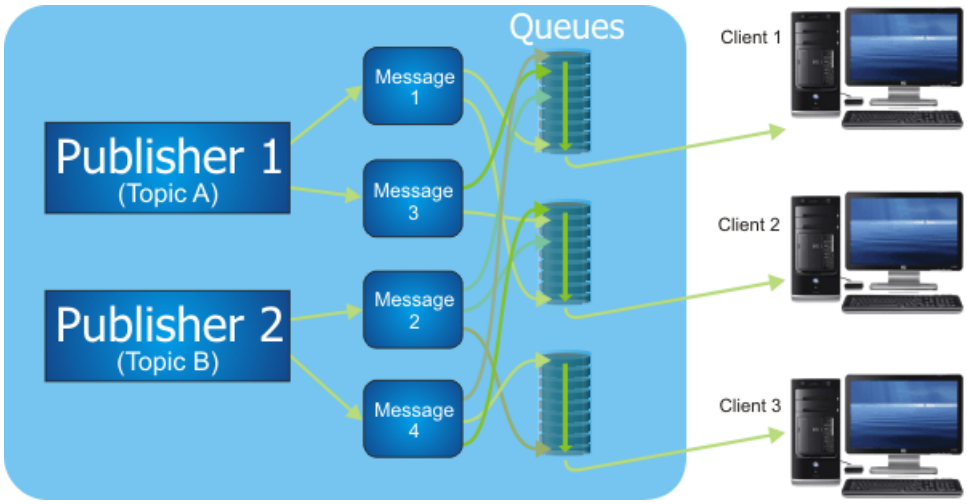


Figure 14: Queues

This shows four Messages being created by two Publishers on two different Topics. The Message numbers indicate the order in which the Publishers actually publish them. Clients 1 and 2 are subscribed to both Topics and so receive all four Messages but Client 3 is subscribed only to Topic B and so only receives those Messages from Publisher 2.

When a Publisher sends a Message it can request that it is high or low priority which will influence the message delivery order. For example, a high priority message will go to the front of the queue.

If duplicate Messages become enqueued for a Client then you may want to prevent duplicates being sent in order to reduce Message traffic. This is known as 'conflation'. You may also want to merge multiple messages for the same Topic into a single Message - this is known as 'structural conflation'.

You may also want to reduce the rate at which Messages are dequeued and sent to certain Clients. This is known as ' Throttling '.

Limits may be set on Queue sizes so that they cannot grow indefinitely and it is possible for Publisher to be notified when Queue sizes reach pre-defined thresholds.

Concurrency

Diffusion™ supports high numbers of concurrent Client connections and concurrent processing of requests.

Diffusion™ utilises the Java NIO framework so that a single Diffusion™ Server can service a very large number of concurrent Client connections without the need for separate threads of processing for each connection. Thread Pools are utilised to process Messages.

As each Message is read it is given to a thread from an 'Inbound thread pool' for processing allowing the NIO thread to accept the next Message.

When a Publisher publishes (or sends) a message it is queued for processing so that the Publisher thread itself does not block. From that queue the Client Multiplexers route the Messages to the appropriate Client queues and also dispatch Messages (batched into efficient output buffers) to the Clients. All formatting for different transports, conflation etc is performed in the Multiplexer threads.

Copyright 2013 Push Technology

Inbound Messages (from Clients, Event Publishers etc) are accepted by the NIO thread and passed to a thread from the Inbound thread pool for processing. This thread enters the Publisher and performs the required processing. When the Publisher enqueues Messages for Clients its controlling Thread unwinds and a Client Multiplexer processes queued Messages and sends them to Clients.

Glossary

A Glossary of Terms

Table 1: A Glossary of Terms

Term	Definition	See
API	Application Programming Interface. Allowing user written applications written in various languages to interface with the Diffusion™ Product.	API's
Client	A Client is an entity that can receive Messages published by Publishers within a Diffusion™ Server on Topics that the Client subscribes to. Typically a Client is a user written application communicating with a Diffusion™ Server via a Client API or the Diffusion™ protocol. A Publisher can be a Client of another Publisher in a distributed environment.	Clients
Connector	A 'Connector' is a configured point of connection to a Diffusion™ Server. There may be one or more connectors (each listening on a different port). Each connector may accept single or multiple types of connection	Connectors
Conflation	Conflation is the facility to prevent transmission of unnecessary duplicate Message information in order to reduce network traffic.	Conflation
Diffusion™	The Diffusion™ product comprising the Diffusion™ Server and APIs.	Website
Event Publisher	An Event Publisher is an application connecting to a Diffusion™ Server using the Event Publisher Protocol to send Messages to the Server. Each Message is routed by the Server to the Publisher owning the Message's Topic. The Event Publisher may be an API or a raw Protocol connection. The purpose of an Event Publisher is to act as a data feed to Publishers.	Event Publishers Java API Windows API
External Client	An External Client is a Client application that connects to a Diffusion™ Server using the External Client Protocol. The application can use an External Client API or the raw Protocol.	Java API Windows API
Fetch	A request from a Client to a Publisher for the current state of a Topic.	Topic Fetch
Message	A Message is a series of bytes of information formatted according to the Diffusion™ protocol which may be sent between components utilising Diffusion™.	Messages
Message Queue	A queue of Messages. Each Client connection to Diffusion™ has such a queue upon which Messages are put for sending to the Client.	Client Queues
Publisher	A Publisher is the component which publishes Messages relating to one or more Topics. A Server may host one or more Publishers. Messages sent by Clients on particular Topics are routed to the Publisher that owns the Topic. Publisher functionality is provided by users by writing a Java Publisher class.	Publishers
Protocol	The Diffusion™ Protocol defines the exact format of Messages passed between Diffusion™ components. Where an API is not available for a particular language, socket based connections may be made to a Diffusion™ Server and the raw protocol used for communication.	Protocol
Server	The Diffusion™ Server is a software component that provides Diffusion™ functionality. Clients may connect to Servers and subscribe to Topics. The	Server

Copyright 2013 Push Technology

Term	Definition	See
	server broadcasts Topic Messages to all subscribed Clients. Servers host Publishers.	
Subscription	The Diffusion™ Server is a software component that provides Diffusion™ functionality. Clients may connect to Servers and subscribe to Topics. The server broadcasts Topic Messages to all subscribed Clients. Servers host Publishers.	Subscription
Throttling	A Topic is a logical channel upon which a Publisher can publish Messages such that any Client subscribed to that Topic will receive those Messages. It is bi-directional in the sense that a Client can send a Message to a Topic which will route that Message to the Publisher providing the Topic.	Throttling
Topic	A Topic is a logical channel upon which a Publisher can publish Messages such that any Client subscribed to that Topic will receive those Messages. It is bi-directional in the sense that a Client can send a Message to a Topic which will route that Message to the Publisher providing the Topic.	Topics
Topic Tree	The total set of Topics at a Server arranged as a hierarchical tree. The Topics of all Publishers at the Server will be in the tree.	Topic Hierarchy
Message	A Message is the unit of transmission of data within Diffusion™. A Message comprises a fixed header, some optional headers and an optional data payload.	Messages

Copyright 2013 Push Technology

Chapter 5

Installation

Topics:

- *System Requirements*
- *Graphical Installation*
- *Headless Installation*
- *Licensing Instructions*
- *Installed Files*
- *Testing your installation*
- *Web Server Installation*

Installing Diffusion™

Diffusion™ is shipped with a Jarball file and an installation guide in HTML format.

There is a file called Diffusion*n.n.n*.jar (where *n.n.n* is the release number) which contains all of the files required to install Diffusion™.

Diffusion™ can be installed either through the graphical installer provided or, where appropriate, using terminal commands (headless mode).

System Requirements

Requirements and Recommendations for Running Diffusion™

Hardware Configuration

Before starting the installation procedure, review the topics in this section to determine that your system meets the basic requirements and that you have the prerequisite software installed.

Diffusion™ is primarily tested on the following hardware:

- Intel Xeon E-Series Processors
- 8Gb RAM
- Intel Gigabit NICs with TCP Offloading and Intel/Solar Flare 10GBE cards for performance testing.

NIC, CPU and RAM (in decreasing order of importance) are the three components that will have the biggest impact on performance: a massively rapid file system is not considered a necessity although 15K SAS drives and RAID 10 are used in-house. Intel hardware is used due to its ubiquity in the marketplace and proven reliability.

Diffusion is also certified on VMWare with the following hardware

Host specifications:

- 2 Dell E5-2650
- 32Gb RAM
- 240GB OCZ Agility 3 SSD
- VMware VSphere 5 Dell release

Virtual machine specifications:

- 8 VCPUs
- 28Gb RAM
- 30GB SAS emulated HDD
- Intel X540 (Non-direct passthrough)
- CentOS 6.3

The results shows the VM delivers a comparable performance.

Note: For best performance, a non virtual installation is recommended.

Operating System

Diffusion™ is certified and tested on the following software:

- Red Hat 6.0
- Centos 6.3
- Windows Server 2008 R2

Linux is strongly encouraged for Diffusion™ installs, due to enhanced performance, and a distribution with enterprise-level support available, such as Red Hat or SUSE Enterprise is recommended.

Java

Diffusion™ is exclusively tested with 64bit Oracle Java 7 distributions (currently 1.7_u45 64bits). Open source equivalents are not supported and it is suggested that all clients add the official JDK via the packet manager of their OS.

Networking

The use of F5 load balancers with SSL offloading is recommended.

Copyright 2013 Push Technology

Graphical Installation

Using the installer UI

1. To install using the graphical installer, double click the `install.jar`. The graphical installer can also be launched from the command line by typing `java -jar install.jar Diffusion`
2. The Installer wizard starts, guiding you through the Diffusion™ installation process. The installer first checks software and system requirements are satisfied for Diffusion™ to run. Specifically, it checks for the presence of a compliant JDK or JRE version as well as a suitable Diffusion™ jar file. The results are summarised in the first screen of the wizard shown below.
3. A confirmation dialog box is displayed.

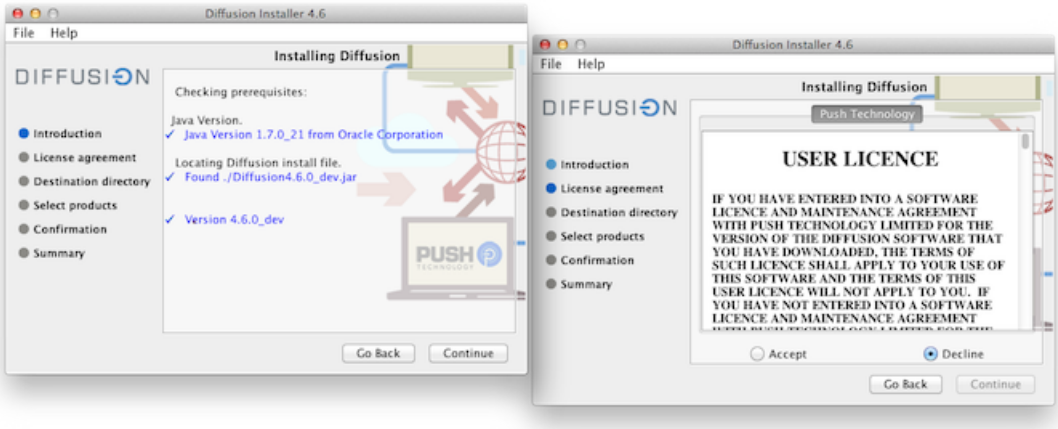

Diffusion Installer 4.6
File Help
Installing Diffusion
DIFFUSION
Checking prerequisites:
Java Version.
Java Version 1.7.0_21 from Oracle Corporation
Locating Diffusion install file.
Found ./Diffusion4.6.0_dev.jar
Version 4.6.0_dev
Introduction
License agreement
Destination directory
Select products
Confirmation
Summary
Go Back
Continue
Push Technology
USER LICENCE
IF YOU HAVE ENTERED INTO A SOFTWARE LICENCE AND MAINTENANCE AGREEMENT WITH PUSH TECHNOLOGY LIMITED FOR THE VERSION OF THE DIFFUSION SOFTWARE THAT YOU HAVE DOWNLOADED, THE TERMS OF SUCH LICENCE SHALL APPLY TO YOUR USE OF THIS SOFTWARE AND THE TERMS OF THIS USER LICENCE WILL NOT APPLY TO YOU. IF YOU HAVE NOT ENTERED INTO A SOFTWARE LICENCE AND MAINTENANCE AGREEMENT
Accept
Decline

Figure 15: Installer first steps

4. You will be prompted to accept the End User License Agreement (EULA) and to enter the installation path for the Diffusion™ files. The second screen above lists the optional extra Diffusion™ products you want to install.
5. Once you choose the modules to install, the installer starts installing Diffusion™ files in the directories specified. The final screen above confirms the installation has been successful and indicates how to start the Diffusion™ server. Note: once the installer is launched, it is possible to get the installer to install the licence file at the same time. To enable this feature, selected 'load licence file' from the file menu.

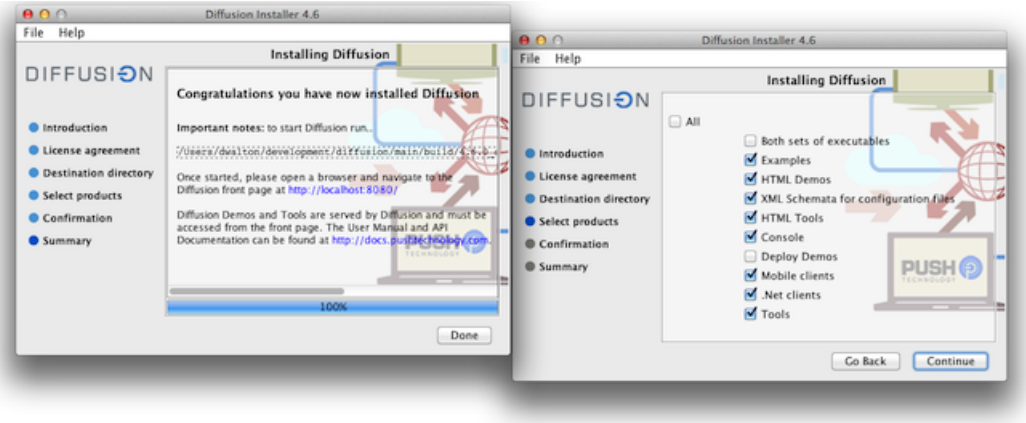


Figure 16: Finishing off

Headless Installation

A guide to installing Diffusion™ from a command line

Copyright 2013 Push Technology

1. Installation in headless mode may be used in circumstances where the (graphical) installer cannot be used or is not appropriate. To install Diffusion™ in a headless environment open a terminal window, change the directory to that where the Diffusion™ jar files are located and type the following line:
2. *java -jar install.jar Diffusion n.n.n*.jar
3. The headless installation (which can be forced using the -Djava.awt.headless flag) follows the same lines as the graphical mode, with the exception that the interaction takes place via the terminal rather than via a graphical widow. Like the graphical installation, you will be prompted for acceptance of EULA and for the installation path. The next step will see the installer asking if all packages should be installed and, in case that is not required, it will ask the installation of all available packages sequentially. The rest of the installation will follow the same process as the graphical installer.

Licensing Instructions

How to install the Diffusion™ licence

1. A Diffusion™ licence is a file with a ".lic" extension. It can be installed either via the graphical installer or manually. In the first case, you can load the Diffusion™ Licence at any time after the installer has been launched. Select the Load licence file option from the File menu and find and select the .lic file. Installation will continue normally, with the licence file being copied to the `etc` directory with the name `licence.lic`. When installation is complete, it will be possible to execute the Diffusion™ server immediately.

Copyright 2013 Push Technology

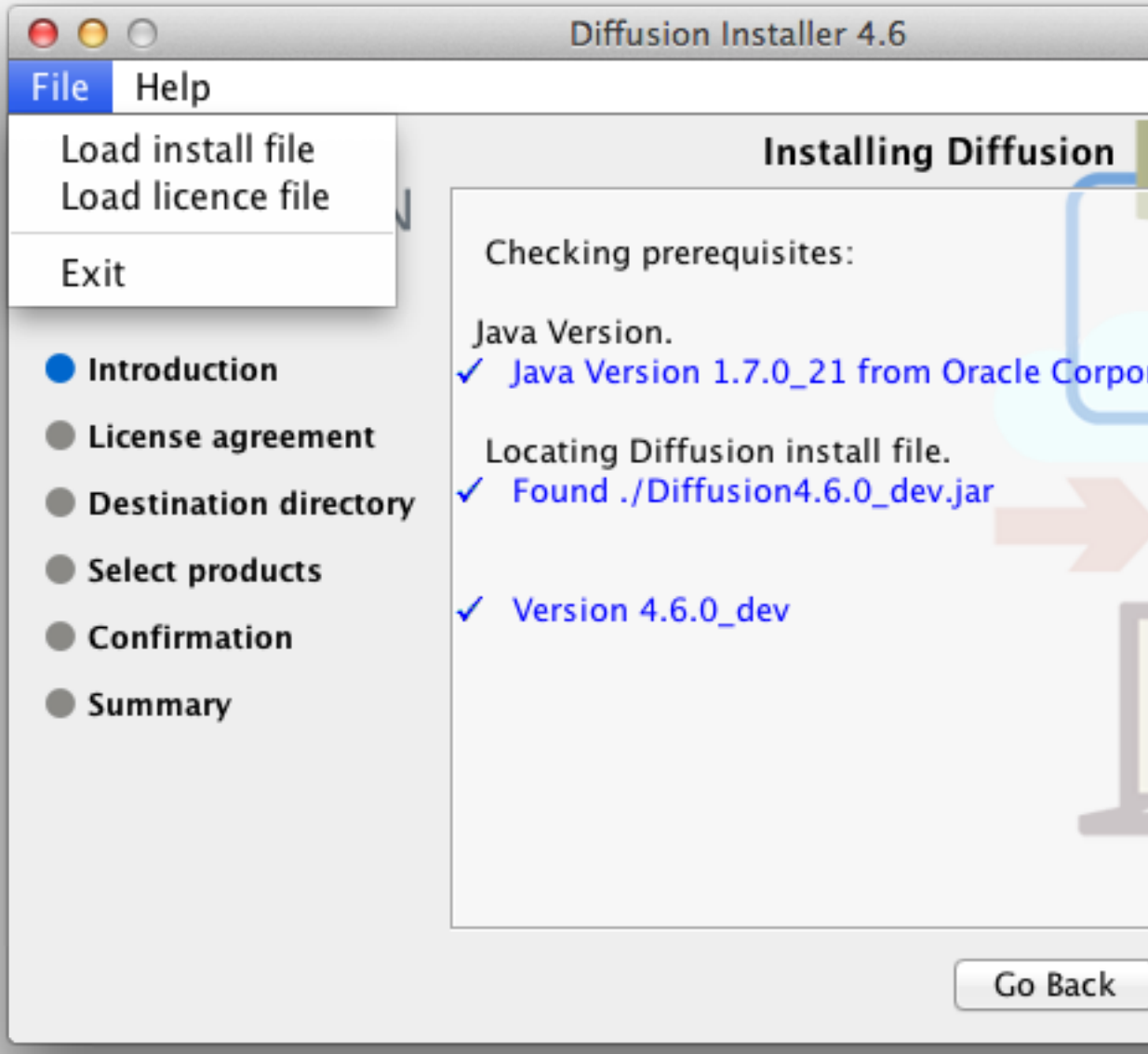


Figure 17: Loading a licence using the installer

2. To install manually, copy the licence file to `licence.lic` in the `etc` directory. You may also keep the original name, although this will require modifying the command used to start Diffusion™.
3. The licence file contains information about which products and components are licensed, the expiry dates for each component, applicable user limits and host network address restrictions.
4. If the distribution component is not licensed, the server will be unable to make outbound connections to other Diffusion™ Servers or to accept inbound connections from other Servers.

Copyright 2013 Push Technology

Installed Files

After installing Diffusion™ the following Directory Structure will exist:-

Table 2: Installed files

Folder Name	Contents
bin	Contains the main Diffusion™ executables for starting Diffusion™ or a Management VM
data	This directory is always on the Server classpath and so may be used for files that needs to be loaded at runtime. The installed directory will contain some files used by issued demo Publishers as well as some example Client Validation Policy files.
docs	Contains all issued documentation including API specifications.
etc	Contains the properties files for configuring Diffusion™. Examples are issued which will work for demos. Example Policy files for Silverlight and Flash are also issued.
examples	Contains example Java Programs including the source of all issued demos.
ext	Contains the external jar files that Diffusion™ needs to run.
html	This contains files required to run Diffusion™ from a web server including all demo examples.
lib	Interoperability libraries. This directory contains libraries for the development of clients on various platforms (e.g. C, Java, Flex , .NET, mobile devices, etc.). This directory also contains the main Diffusion jar file upon which the Server relies.
obj-release	Contains the .NET dlls for the Windows APIs.
logs	A place holder for Diffusion™ and web server logs.
stresstest	Contains the stress test package.
tools	Tools and Utilities that help with testing and deploying Diffusion™ (see below).

Tools & Utilities

The following describes the contents of the issued tools directory. Note that the files present and their suffixes may vary according to the platform that the product is installed on.

Table 3: Tools and Utilities

DiffusionEventPublisherTest.exe	Windows Event Publisher Test Tool.
DiffusionExternalClientTest.exe	Windows External Client Test Tool.
eventpub.bat/sh	Generic Event Publisher Test Tool.
extclient.bat/sh	Generic External Client Test Tool.
externalclienttest.properties	External Client test tool properties.
war.xml	Example war.xml file
web.xml	Example web.xml file

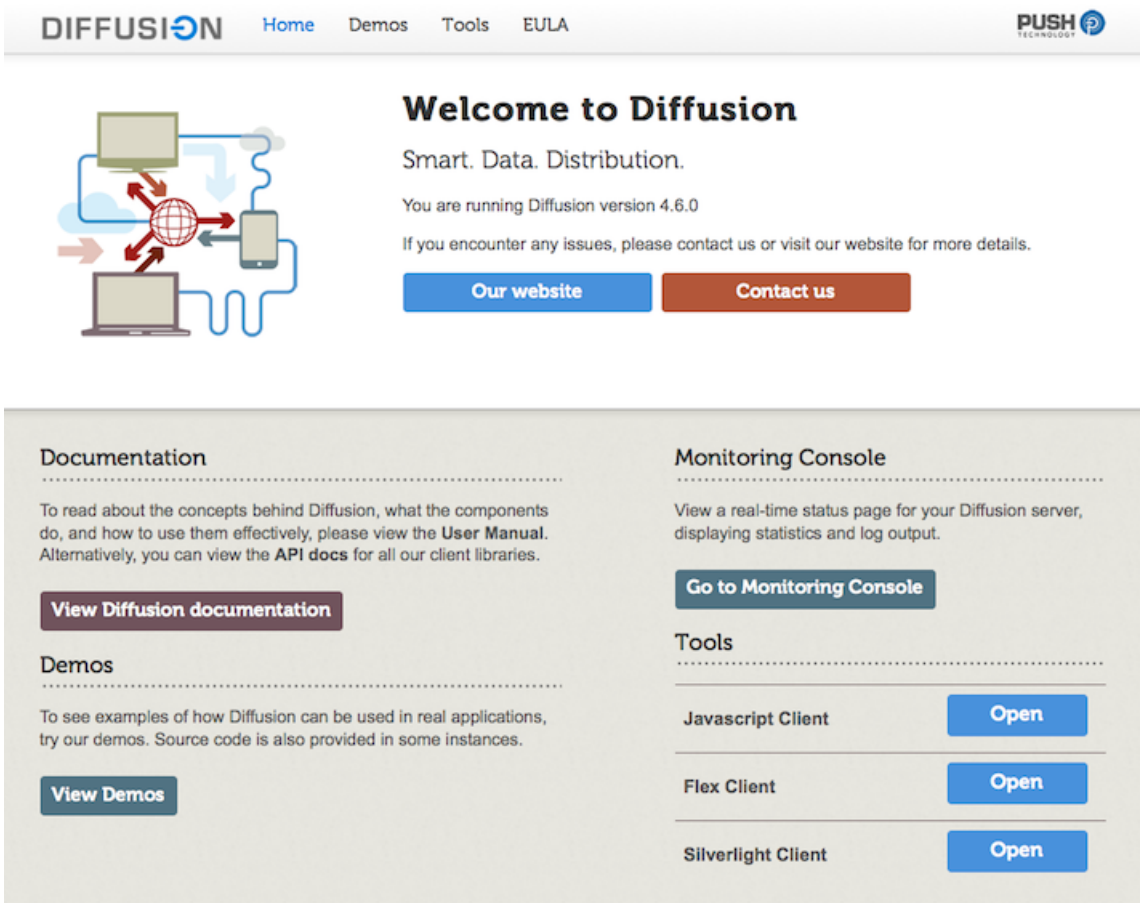
Copyright 2013 Push Technology

init.d	contains an example init.d file to start Diffusion™ as daemon on MacOS/ Linux/UNIX systems.
jstack.sh	Prints stack traces for threads within a Java VM. Based on the Java jstack tool.
nagios	Integration files for the Nagios IT Infrastructure Monitoring Tools.

Testing your installation

Testing the Diffusion™ installation

1. After installation, all of the Diffusion™ files will be available in the directory specified during installation. Start the Diffusion™ server using +one of the scripts contained in the bin directory created during installation.
2. On windows machines, start the server by executing the diffusion.bat script file. On MacOS/Linux/UNIX machines, the server can be started by executing the diffusion.sh file.
3. As the server boots, and modules are executed, the terminal window will display logging information as to the status of execution. The user who wants to check for successful server execution should look for a log message containing: `INFO|main|PUSH0165|Diffusion Server started.|com.pushtechnology.diffusion.DiffusionController`
4. This line is typically the last one to be printed on terminal. Inspect the entire execution log printed in the terminal to search for `WARN` messages to ensure that all components have started correctly. Once the server is up and running, opening a browser to `http://<serverAddress>:8080` (or `http://localhost:8080`) will show the Diffusion™ Landing Page. At this point, the Diffusion™ server is ready to be used.


DIFFUSION
Home
Demos
Tools
EULA
PUSH
Welcome to Diffusion
Smart. Data. Distribution.
You are running Diffusion version 4.6.0
If you encounter any issues, please contact us or visit our website for more details.
Our website
Contact us
Documentation
To read about the concepts behind Diffusion, what the components do, and how to use them effectively, please view the User Manual. Alternatively, you can view the API docs for all our client libraries.
View Diffusion documentation
Demos
To see examples of how Diffusion can be used in real applications, try our demos. Source code is also provided in some instances.
View Demos
Monitoring Console
View a real-time status page for your Diffusion server, displaying statistics and log output.
Go to Monitoring Console
Tools
Javascript Client
Open
Flex Client
Open
Silverlight Client
Open

Copyright 2013 Push Technology

5. The Landing page provides pointers to sources of information regarding legal terms and conditions (e.g. EULA), user guides, API documentation and demos. After installation, the issued files will be in the specified install directory. The Diffusion™ product will work as issued with an assortment of demo Publishers that may be tried out to ensure that the installation functions as it should. To test the installation, using a browser navigate to `http://server-ip:8080` This will load the Diffusion™ Landing Page and *the demos* (if installed) can be accessed from there.

   Web Server Instructions

   By Default, Diffusion™ is a Web Server if you have configured a HTTP connector to do so, but if you want to run Diffusion™ along side a Web Server then see the section on Web Server Installation - Testing the Installation. After installation the issued files will be in the specified install directory. The product will work as issued with an assortment of demo Publishers which may be used to ensure that the installation is functioning properly.

Web Server Installation

Diffusion™ can act as a Web Server by modifying the `etc/Connectors.xml` and adding a web-server definition to a connector. If you wish to use another Web Server to as your Web Server for Diffusion™ then further configuration is required. All requests at context /diffusion will need to be forwarded to the port that the HTTP requests are listening on, which by default is 8080. The default ports of Tomcat and the HTTP connector are the same. Only one of these can bind the address port. Diffusion™ can be started via a Servlet request and can run within any Servlet container. Examples below:-

Servlet Configuration

This will explain how to setup a servlet for Diffusion™. You may find it eaiser to create and deploy a web application using the ant script discussed below. The following lines need to be added to `WEB-INF/web.xml`. The `servlet` section needs to go with the other `servlet` sections. A `servlet-mapping` entry is not needed for the `DiffusionServlet` as it does not handle any requests.

```
<servlet>
    <servlet-name>Diffusion</servlet-name>
    <display-name>Diffusion Servlet</display-name>
    <servlet-class>com.pushtechnology.diffusion.servlet.DiffusionServlet</
servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
```

Make sure that the JAR files in `ext` are in the lib directory of `WEB-INF`. Also copy all of the properties files from the `etc` directory to the `classes` directory. Make sure that the policy files are correctly pathed in the *`etc/Connectors.xml`* file. The `html/lib/DIFFUSION` directory will also need to be copied into the `webapp` directory as `lib/DIFFUSION`, this directory contains the browser API libraries.

Creating a Web application archive

In the `tools` directory of the default installation is an Ant build file `war.xml` that will package the Diffusion™ into a WAR file that you can deploy to a servlet container. Invoking the `war` or `all` target of the file will build the archive at `build/diffusion.war`. The diffusion.war file can then be deployed to your servlet container according to its documentation.

A WAR file is an archive like a JAR that contains a WEB-INF directory. This directory contains XML based configuration describing the web application and the directories classes and lib that contain .class or .jar files that will be added to the classpath. The top level of the WAR file contains resources that can be served by Tomcat. The configuration files for Diffusion should be placed in the WEB-INF/classes directory. The diffusion.jar and files from the ext directory should be placed in the WEB-INF/lib directory.

Tomcat Configuration

Copyright 2013 Push Technology

Diffusion uses the java.util.prefs functionality so Tomcat must be started with the userRoot configured. Without this warning messages will be generated in the logs. The tomcat user wil require write permissions to the userRoot directory.

```
-Djava.util.prefs.userRoot=/var/lib/tomcat6/diffusion/prefs/user
```

Tomcat must have connectors defined to handle incomming connections. A connector defines the port, protocol and various properties of how the connection should be handled. Connectors are defined in the `Server.xml` file and you should refer to Tomcat documentation for more information. The following is an example connector for handling HTTP 1.1 connections on port 8080.

```
<Connector port="8080"
        connectionTimeout="20000"
        URIEncoding="UTF-8"
        maxThreads="3"
        protocol="HTTP/1.1" />
```

Accessing Publishers from Tomcat

Diffusion started within Tomcat allows Tomcat to access the Publishers. Tomcat can be used to serve JSP files providing dynamically generated content. These files can access publisers using the Publishers class static methods.

```
<%@ page import="java.util.List,com.pushtechnology.api.publisher.*" %>
<html>
    <head>
        <title>Publisher Information</title>
    </head>
    <body>
        <table>
            <tr>
                <th>Publisher Name</th>
                <th>Topics</th>
            </tr>
            <% for (Publisher pub : Publishers.getPublishers()) { %>
            <tr>
                <td><%= pub.getPublisherName() %></td>
                <td><%= pub.getNumberOfTopics() %></td>
            </tr>
            <% } %>
        </table>
    </body>
</html>
```

The above is the content of a JSP file that will return a list of the Publisher Diffusion is running with the number of topics each publisher owns.

Apache Mod Proxy installation

Apache Mod Proxy can be used to forward HTTP requests from an Apache webserver to Diffusion. It does not support persistent connections or websockets so the Websocket and HTTPC connections will not work. Make sure that you include the following into the Apache config file (Virtual host setting).

```
ProxyPass /diffusion/ http://localhost:8080/diffusion/
```

For more infomation you should refer to the Apache Mod Proxy documentation.

Apache AJP13 Installation

Apache AJP can be used to forward requests from an Apache webserver to Tomcat. In the Apache virtual host configuration, mount the path

```
JkMount /diffusion/*dfnjetty
```

Copyright 2013 Push Technology

Workers definition file

```
worker.dfnjetty.port=8009
worker.dfnjetty.host=(host IP)
worker.dfnjetty.type=ajp13
worker.dfnjetty.lbfactor=1
worker.dfnjetty.cachesize=50
worker.dfnjetty.socket_keepalive=1

worker.list=dfnjetty
```

A connector that handles the AJP/1.3 protocol is needed running on port 8009 (because of the Workers file desribed above). You should refer to the Tomcat documentation for more information on this.

IIS Installation

You need to an ISAPI_Rewrite tool. For example *http://www.helicontech.com/isapi_rewrite*

The re-write rule is as follows:-

```
RewriteEngine on RewriteRule ^diffusion/ http://localhost:8080/diffusion/
 [p]
```

Licensing

The Diffusion™ license needs to be loaded by a Java agent. This Java agent must be added to the Java VM arguments that are used to start the servlet container. Two options must be passed to the agent: the license file and a public key store. How to set up the servlet container you are using will vary between containers, operating systems and configurations. You should consult the documentation for the container you are using. The argument you need to add should look similar to:

`-javaagent:<PATH>/licenceagent.jar=<PATH>/licence.lic,<PATH>/publicKeys.store`

The licenseagent.jar contains the Java agent and is found in the `lib` directory of the default installation. The key store file is included in the `etc` directory of the default installation. As these files need to be referenced when the container starts they are not a part of the Diffusion™ servlet. The paths will need to be absolute or relative to the servlet container, not the web application.

Diffusion Configuration

The built-in Diffusion web server is configured using the *WebServer.xml* file. When using a third party web server at least some of this functionality can be disabled. The file-service, and two http-service entries can be removed as Tomcat will provide this functionality. The client-service is needed to support WebSocket, HTTPC and HTTP connection protocols. If these are not used it can be disabled as well.

Copyright 2013 Push Technology

Chapter 6

Server

Topics:

- *Server Basics*
- *Starting the Server*
- *Running from Java Application*
- *Concurrency*
- *Connectors*
- *Load Balancers*

The Diffusion™ Server is the core component of the Diffusion™ Product.

The Server is written in Java and therefore can be deployed on any platform that supports Java.

See the *Installation* section for details of how to install a Server instance.

Server Basics

What is the Diffusion™ Server?

The Diffusion™ Server is a highly efficient and low latency messaging server which provides the functions listed here.

A scalable and mature solution to push (stream) and receive data and events, in real-time, both to and from a web browser and other 'net' connected devices.

- A scalable and mature solution to push (stream) and receive data and events, in real-time, both to and from a web browser and other 'net' connected devices.
- High performance transport of data between two or more machines within your own network or 'extranet'.
- Web Server and Media Server Capability.

A single Server may be deployed to push messages to web browsers (or other net devices) or many interconnected Servers can provide a fully scalable enterprise messaging solution.

Starting the Server

Here we learn the basic of how to configure and start a Diffusion™ Server.

A Diffusion™ Server instance may be installed as described in the *Installation* section. Once installed the installed files will be present in the specified install directory. As issued, a Diffusion™ Server will be configured to run the demo Publishers and these may be tried out and explored to familiarise with the product.

However, for a real implementation it would first be necessary to design and write Publisher and Client applications.

Once a publisher is made available then Diffusion™ may be configured to run it. This is done by editing the XML property files - see the *Configuration section* for more details.

To configure the Server to run a user written Publisher would involve the following:-

1. Define Connectors appropriate for the types of connections (Client, Event Publisher, etc) that will be supported using `etc/Connectors.xml`
2. Define Publishers using `etc/Publishers.xml` .

   Other configuration may be required according to the nature of the application. The above steps would be the bare minimum requirement to get a Publisher working.

   The Server produces logs which record actions performed by the Server and may also be used for diagnostic purposes. The level of logging produced and where logging is directed is configurable. See *Logging* for further details.
3. The `diffusion.sh` or `diffusion.bat` command (issued in the `bin` directory) will start Diffusion™. An optional properties directory can be specified as a parameter to be used instead of the default `../etc` directory.

   Once a Server is started then it should be accessible from Client applications or Test Tools.
4. A running Server may be managed using JMX or via the Management API
5. Tuning the Server is critical to getting the best performance out of a Diffusion™ installation.

   Some of the aspects that need to be considered are Message Design and Sizing, Concurrency, Client Queues and Connection Buffers. All of these issues are discussed in detail in the *'Tuning' section*.

Running from Java Application

Running a Diffusion™ Server from within a Java Application

Copyright 2013 Push Technology

Creating a Server

In order to run Diffusion™ from within a Java application a DiffusionServer object needs to be instantiated, configured and then started.

DiffusionServer is available in the api.server package and may be instantiated as follows:-

```
DiffusionServer server = new DiffusionServer();
```

If done in this way then all required aspects of the server must be configured programmatically before it is started.

Alternatively, when creating the Server, the name of a directory from which to load XML property files may be specified. This will cause any XML files in that directory to be loaded into configuration objects which can form a basis for additional programmatic configuration. A full set of files may be present and then tuned as required or just a partial set of the files may be present and all missing configuration then supplied programmatically.

```
DiffusionServer server = new DiffusionServer(directoryPath);
```

Configuring the Server

Once the Server object has been instantiated it may then be configured. The root configuration object may be onbtained from the server object as follows:-

```
ServerConfig config = server.getConfig();
```

Alternatively the root could be obtained using `ConfigManager,getServerConfig()`.

Most configuration properties have sensible defaults but some configuration objects need to be be supplied for a server to function. For example:-

- **Publishers** - Unless all Publishers are to be hot deployed then at least one Publisher would need to be defined.
- **Connectors** - At least one connector would need to be defined for clients to connect to. If no connector is defined then at startup a default connector will be created and a warning logged.
- **Queues** - Definitions of Client queues need to be provided. If none are supplied then defaults will be set up on startup and a warning logged.
- **Multiplexers** - At least one multiplexer defintion needs to be configured. If none are supplied then a default one will be created at startup and a warning logged.
- **Thread Pools** - Thread Pool definitions are required to specify the characteristics of the thread pools the server uses. If none are supplied then a default defintion will be created on startup and a warning will be logged.

So a typical minimum configuration set up might be:-

```
DiffusionServer server = new DiffusionServer();

ServerConfig config = server.getConfig();

// Publisher
PublisherConfig publisher = config.addPublisher("My
 Publisher","com.company.MyPublisherClass");

// Connector ConnectorConfig connector = config.addConnector("Client
 Connector");
// Configure connector as required....

// Thread Pools
ThreadsConfig threads = config.getThreads();
ThreadPoolConfig inbound = threads.addPool("Inbound");
inbound.setCoreSize(3);
inbound.setMaximumSize(10);
inbound.setQueueSize(2000);
inbound.setPriority(8);
```

Copyright 2013 Push Technology

```
threads.setInboundPool(inbound.getName());
threads.setBackgroundPoolSize(2);

// Queues QueuesConfig queues = config.getQueues();
QueueConfig queue = queues.addQueue("DefaultQueue");
queue.setMaximumDepth(10000);
queues.setDefaultQueue("DefaultQueue");

// Multiplexer MultiplexerConfig multiplexer =
 config.addMultiplexer("Multiplexer");
multiplexer.setSize(4);
```

Starting the Server

Once the Server configuration has been completed then the server may be started using `server.start()`.

The delclared Publishers will then be loaded and connectors will start to listen on the configured ports.

Stopping the Server

The server may be stopped using `server.stop()` at which point it will no longer be available.

Run Requirements

The Diffusion™ Server application must be run from within the `bin` folder in the Diffusion™ install folder (or a folder at the same level) for all of the relative paths in the configuration to work properly and for Diffusion™ to be able to locate the runtime jars it requires. The application could be run from elsewhere but this would involve changing a number of configuration items that specify relative paths and also adding jars from the `ext` directory to the application classpath.

In order to run a Diffusion™ Server the following items from within the Diffusion™ install directory need to be added to the classpath for the VM:-

- `../lib/diffusion.jar`
- `../etc`

The `etc` folder is optional. It would only be required if base XML configuration files were to be loaded from it. Alternatively you could add any other directory containing configuration files.

Additionally, the functioning of the Diffusion™ Server relies upon the use of a Java agent to facilitate licence checking. The java command that starts the server application must therefore include the following argument:-

`-javaagent:../lib/licenceagent.jar=../etc/licence.lic,../etc/publicKeys.store`

So an example unix script to run in the `bin` directory might be:-

```
CP=../lib/diffusion.jar:../etc
java -javaagent:../lib/licenceagent.jar=../etc/licence.lic,../etc/
publicKeys.store -cp ${CP} com.mycompany.MyApp
```

Limitations

Currently only one Diffusion™ server may be instantiated in a Java VM and may only be started once.

Concurrency

Diffusion™ is a multi-threaded Server and utilises concurrent processing to achieve maximum performance. Java NIO technology is utilised so that a separate thread is not required for each concurrent connection therefore very large numbers of concurrent connections can be handled.

Copyright 2013 Push Technology

Because Diffusion™ is a multi-threaded environment it is necessary to have an understanding of concurrency issues when writing Publishers and when configuring Diffusion™ for best performance (see *Tuning*).

This section discusses issues of threading and concurrent processing.

Publisher Threads

The processing that occurs within the user written code of a Publisher may be executed in different threads as discussed below. It is therefore possible for any Publisher method to be called at the same time as another. Because of this all Publisher processing must be threadsafe and it is the user's responsibility to synchronize processing as required. It is recommended that synchronization is maintained at the smallest scope possible to avoid performance bottlenecks.

Inbound Threads

Any input that is received on an NIO connection is processed by a thread from the Inbound Thread Pool . This includes most Publisher notifications from Clients, Event Publishers or other Publishers with the exception of control notifications (such as `initialLoad` , `publisherStarted` etc) which would occur in the controlling thread.

It should be noted that the act of Publishing or sending Messages to Clients is asynchronous that is to say that the Message is queued for the Client(s). Publisher processing is not blocked whilst Messages are delivered to Clients. For best performance it is recommended that any code executed in the inbound threads is non-blocking (i.e avoid database access, locking, disk IO etc as much as possible).

Client Notification Threads

If a Publisher uses Client Notifications then the Publisher will have its own dedicated thread to process those notifications.

By default here is one notification thread per Publisher, no matter how many listeners are defined. Each event is processed by the thread in the order in which they occur and therefore two Client Notification event methods will not be called concurrently. If the order of such events is not critical then it is possible to specify that a user thread pool is used for client notifications this increasing throughput.

User Threads

Publishers or other users of the Java API may make use of the Java Threads API to schedule tasks for processing of their own in a separate thread of processing.

It is possible to execute any object of a class that implements the `RunnableTask` interface using one of the `ThreadService.schedule` methods. It is possible to request a one-off execution of a task, periodic execution at a given interval or execution according to a 'cron' like schedule. Periodic processing may be important to Publishers that need to 'pull' data updates from elsewhere.

Such tasks issued via the thread service are executed using threads from the Background Thread Pool.

Alternatively, users may define their own thread pools to use using the Thread service and execute tasks using these thread pools.

NIO Threads

Each ' Connector ' that is configured in *`etc/Connectors.xml`* will comprise a 'Connector' thread and one or more 'Acceptor' threads. The 'Connector' thread listens for incoming socket connections, accepts them and then registers them with an 'Acceptor' thread which will handle any incoming data notifications. Message decoding, routing to Publishers and appropriate Publisher callbacks are all run in the inbound thread pool. Connector and Acceptor threads are therefore occupied for the minimum amount of time and are completely non-blocking.

Though performance can be improved in extreme case by adjusting the numbers of these NIO threads, no significant processing occurs within them.

Copyright 2013 Push Technology

Client Multiplexers

A 'Client Multiplexer' is a separate thread which is responsible for processing Messages on the Publisher Event queue, queuing for clients (conflating if necessary), taking Messages from Client queues and sending them to the Client(s). A number of these Multiplexers may be configured in order to improve concurrent processing when there are a large number of Clients.

Multiplexers will typically 'batch' these output Messages into output buffers according to the output buffer size configured for the Client Connectors.

Thread Pools

Diffusion™ maintains a number of configurable Thread Pools which are used for a number of purposes

The detailed configuration and tuning of these pools is discussed in the *tuning section*. Thread pools can also be accessed programatically using the `ThreadService` class of Diffusion server API, you should refer to the javadoc for more information on this.

The various types of thread pools are as follows:-

Inbound Thread Pool

This is used to obtain a thread to process any inbound Message received on an NIO connection. For example, Clients and Event Publishers. The maximum number of threads configured for this Pool therefore needs to cater for the maximum required concurrency for incoming requests.

Diffusion™ does not maintain a separate thread for each client connection but rather passes each inbound request from a connection to the inbound thread pool for processing.

For example, when a Client subscribes, the input processing happens on an Inbound thread from the pool, the subscribe method and Topic Loader methods will be run in one of these threads.

Connector Inbound Thread Pools

Individual connectors may configure their own separate Inbound Thread pool to override the use of the default. This may be required if you want different behaviours for each connector or if there are a lot of connectors then due to locking on the inbound thread pool, it is more performant for each connector to have its own inbound thread pool.

Background Thread Pool

The background thread pool is used for executing scheduled tasks. These tasks may be issued by Diffusion™ itself or using a Publisher using the Java threads API.

Diffusion™ will use scheduled tasks for various reasons such as:-

1. Timing out ACK mesages. A scheduled task would execute if a message sent to a client is not acknowledged within its required timeout period.
2. Retrying connections. If a Diffusion™ server cannot connect to another server and there is a retry policy then a scheduled task will be used to retry the connection.

If any publisher uses a lot of scheduled tasks, then the number of threads in this pool may need increasing otherwise waiting tasks may queue.

Unlike other types of pool once the specified number of threads are in use tasks will be queued in an unbounded queue.

User Thread Pools

Within the the Java threads API it is possible for the user to define thread pools that may be used for multi-threaded processing.

Copyright 2013 Push Technology

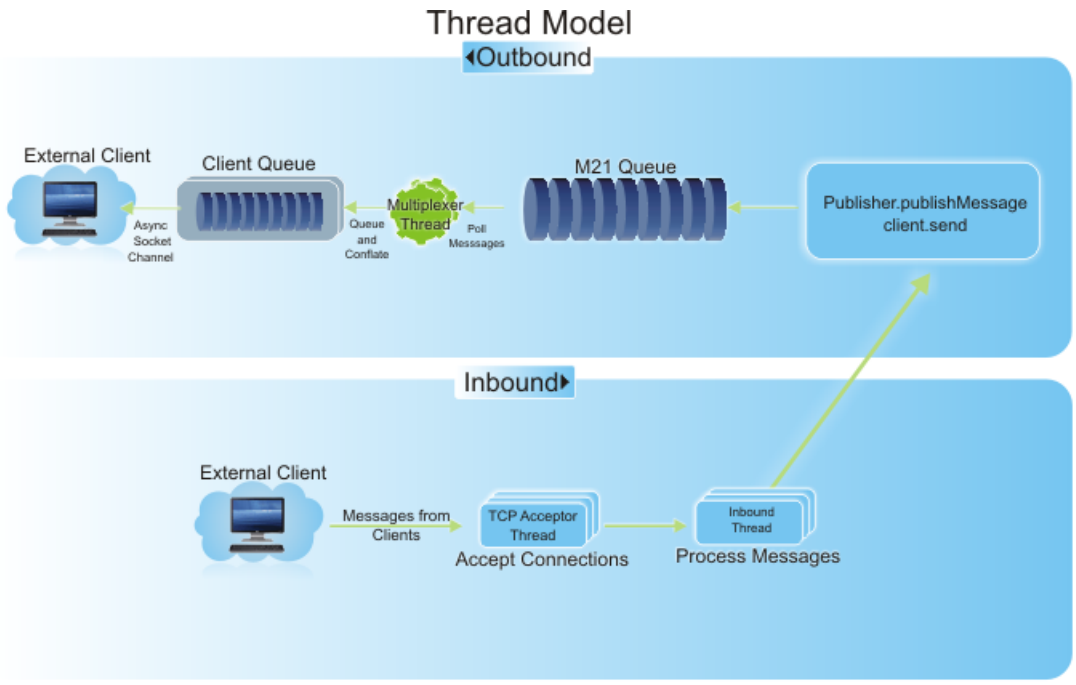


Figure 18: Thread Diagram

Connectors

An introduction to the concept of 'Connectors' within a Diffusion™ Server.

A 'Connector' provides a connection point for external applications to connect to a Diffusion™ Server over a TCP connection.

Each Connector has a 'Socket Server' thread which reacts to an incoming connection and passes it on to an Acceptor to accept and process the connection. The socket information is defined by the Connector.

Suitable Connectors should be defined for inbound connections expected by a Server. Connectors are defined in the `etc/Connectors.xml` configuration file.

The following properties are common to all Connectors:-

Table 4: Connectors properties

Name	A name by which the Connector may be identified which will also be used for its Acceptors.
Port	A Port number on which to accept requests (or Policy Requests).
Host	The host to accept requests (only relevant on a multi-homed machine).
Number of Acceptor Threads	The number of Acceptor threads. May be tuned for performance reasons.
Input Buffer Size	The size of the Socket input buffer to use for each connection.
Output Buffer Size	The size of the Socket output buffer to use for each connection.

Socket buffer sizes are very important in achieving the best performance. See *Tuning* for more details.

Copyright 2013 Push Technology

Restricting Connection Types

By default a Connector can accept any type of connection handled by Diffusion™ but it is possible to configure a Connector so that it will accept only one type of Connection as follows:

Table 5: Connection restrictions

client	Client connections.
policy	Policy File Requests.
event	'Event Publisher' connections.
all	Any type of connection.

Client Connections

Connectors can accept connections from any type of Client . Any number of Connectors can be defined in order to provide different connection points with different properties.

Each Client connection will have an input buffer to receive Messages from the Client. The configured input buffer size must therefore be large enough to accommodate the largest Message expected from the Client.

The output buffer size is used to assign an output buffer per Client Multiplexer into which messages are dequeued prior to transmission. The output buffer size must be at least as large as the largest Message that will be sent to the Client (typically the largest Initial Topic Load Message). However, a much bigger size may be configured so that Messages can be batched for sending to the Client. Ideally the buffer size should be a multiple of the average Message size to allow for suitable batching. This can have an important effect on performance - see the Tuning section for more details. Note that the buffer size must allow extra for header information when using HTTP.

Policy Connections

Connectors are used to serve Policy File requests to plugin Clients. Flash and Silverlight require Policy Files to be served from different ports (Flash on 843 and Silverlight on 943) so if plugin Clients are in use it will be necessary to define a separate Connector for each type of plugin Client.

The Connector configuration specifies the path of an XML Policy file to be sent to the Client.

Event Publisher Connections

The input buffer size specified for an Event Publisher Connector is used to assign an input buffer in the same way as a Client Connector. It must therefore be at least as large as the largest Message expected.

The input buffer size specified may be much larger than the largest expected Message size to allow for buffering and should match the size of the output buffer used by the Event Publisher itself.

The output buffer size specified should be able to accommodate the largest Message that will be sent to the Event Publisher (if any).

Load Balancers

Using Diffusion™ with Load Balancers

Diffusion is commonly used with load balancers such as F5 Networks Inc.'s BIG-IP suite, to distribute incoming requests fairly over a number of Diffusion servers.

Each client is registered against a single Diffusion instance which contains client that is not replicated to other instances. For this reason, it is necessary for all traffic to and from that client is routed to the correct Diffusion server.

For HTTP-based protocols there may be many connections established, dropped and re-established during the lifetime of a single client session. Every one of these must be routed to the correct Diffusion server.

Copyright 2013 Push Technology

At first glance, streaming protocols such as DPT that open a single socket and remain connected until they are no longer required would appear immune to requiring any special considerations. However, in the event that connection keep-alive is enabled to handle reconnections in case of temporary connection loss, it is important that the reconnection attempt is routed to the original server.

HTTP protocols

To route HTTP traffic, the load balancer needs to be able to inspect the HTTP headers and extract session information. Diffusion sets an HTTP cookie named `session` with a connection-specific ID specifically for this purpose. Load balancers typically have facilities for using this to maintain a table of client to server mappings.

Sample HTTP conversation, cookie highlighted:

```
POST /diffusion/ HTTP/1.1
Host: localhost:8080
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:22.0) Gecko/20100101
 Firefox/22.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-gb,en;q=0.5
Accept-Encoding: gzip, deflate
DNT: 1
m: 0
ty: B
t: null
tt: 90
v: 4
username: null
password: null
Referer: http://localhost:8080/tools/DhtmlClient.html
Content-Length: 0
Content-Type: text/plain; charset=UTF-8
Cookie: GUID=95UAaOVCK6FPFB2J9dBl
Connection: keep-alive
Pragma: no-cache
Cache-Control: no-cache

HTTP/1.1 200 OK
Set-Cookie: session=clyde-r6t80mwyjav6
Cache-Control:no-store, no-cache
Content-Type:text/plain; charset=UTF-8
Content-Length:26

4.100.4.clyde-r6t80mwyjav6

POST /diffusion/ HTTP/1.1
Host: localhost:8080
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:22.0) Gecko/20100101
 Firefox/22.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-gb,en;q=0.5
Accept-Encoding: gzip, deflate
DNT: 1
m: 1
c: clyde-r6t80mwyjav6
Referer: http://localhost:8080/tools/DhtmlClient.html
Content-Length: 0
Content-Type: text/plain; charset=UTF-8
Cookie: session=clyde-r6t80mwyjav6; GUID=95UAaOVCK6FPFB2J9dBl
Connection: keep-alive
Pragma: no-cache
Cache-Control: no-cache
```

Copyright 2013 Push Technology

Streaming Protocols

Without the ability to parse headers (and indeed, the absence of a session ID at all), the most common method for routing a streaming protocol such as DPT or websockets is to create a client/server mapping based on the IP addresses of the endpoints. This technique is generally referred to as "Sticky-IP", and has the advantage of also working with HTTP transports, if required.

For F5's Sticky IP, ensure that the Source Address Translation option is set to "Auto Map".

Figure 19: Sticky-IP in F5 BIG-IP

The drawback of this approach is that multiple users masquerading behind a proxy or access point may have the same IP address, and all requests from clients with that IP address will be routed to the same Diffusion instance. Load balancing will still occur, but some hosts may be unfairly loaded.

Routing Strategies

Load balancers often present several different strategies for choosing which server will be associated with a new request. Common examples are:

Table 6: Routing strategies

Name	Description
Round-robin	Each available Diffusion instance is chosen in turn, with none favoured.
Fewest clients	The server with the fewest number of client connections in progress is chosen.
Least loaded	The server with the lowest CPU load is chosen.

Monitors

Determining the availability of a Diffusion node can be achieved in more than one way. Commonly, an HTTP probe is used against the built-in web server. This has the advantage of being simple; most system administrators are familiar with HTTP requests. In the simplest case, a `GET` request can be made against the root context of the web server, for example:

```
GET / HTTP/1.0\r\n
```

However, this only tests the availability of the Diffusion server as a whole, and not the applications within it. A more advanced approach would be to implement an `http-service`, as documented in the *web-servers* section. This would query the state of the Publishers or the Topic tree, and return availability on receipt of a `GET` request.

A better, but more complex approach would be to implement a custom monitor using a scripting language that is supported by the load balancer. In the case of BIG-IP, this would be a custom Diffusion client written in Perl that connects and subscribes to a "status" topic. Details on the Diffusion protocol can be found in the *Protocol* chapter. Note that Push Technology does not provide or support custom monitors written by third parties.

Common Issues

Connection Pooling

Copyright 2013 Push Technology

Many load balancers include a connection pooling feature, such as the BIG-IP OneConnect profile, where connections between the load balancer and the Diffusion server are kept alive and reused by other clients. In fact, multiple clients can be multiplexed through a single server side connection.

In Diffusion, a client is associated with a single TCP/HTTP connection for the lifetime of that connection. If a Diffusion server closes a client, the connection is also closed. Diffusion makes no distinction between a single client connection and a multiplexed connection, so when a client sharing a multiplexed connection closes, the connection between the load balancer and Diffusion is closed, and subsequently all of the client-side connections multiplexed through that server-side connection are closed.

For this reason, it is required that load balancers are not configured to pool connections when working with Diffusion.

Further Reading

F5/BIG-IP:*Enabling Session Persistence*

Copyright 2013 Push Technology

Copyright 2013 Push Technology

Chapter 7

Web Server

Topics:

- *Interaction with Publishers*
- *Security*

Diffusion™ incorporates its own Web Server.

Any Diffusion™ Connector can be configured to act as a Web server and service Diffusion™ Clients, file requests or any request for which a user defined HTTP service has been confgured.

Interaction with Publishers

Describes how Publishers can interact with the Web Server

Server Side processing

Server side processing can be utilised with any file that has a text mime type and Javascript. Currently there are three server side tags, Include, Publisher and Topic Data. These tags are stored in HTML comments so as to not interfere with normal HTML

Include Tag

'Include' stubs load the file specified in the file attribute and are loaded as is into the parent HTML document. They do not necessarily have to be valid HTML. They can be positioned anywhere within the HTML file.

These includes are synonymous with `#Include` statements of ANSI C.

Below is and example of the syntax

```
<!--@DiffusionTag type="Include" file="stub.html" -->
```

Include files can be nested so an include file can contain an include tag

Publisher Tag

Publisher tags allow a publisher to interact with the web page during the serving process. Again these tags can appear anywhere within the HTML document. In the case below the publisher method `processHTMLTag` of the `Trade` Publisher will be called with the tag argument of `table` The Publisher can then return a String of HTML that will be inserted into the document at the position of the Tag and then the Tag is removed. The `processHTMLTag` method is also called with the HTTP Request, although the request cannot be written to. Below is an example of the syntax

```
<!--@DiffusionTag type="publisher" publisher="Trade" tagid="table" -->
```

TopicData

Topic Data tags allow for `SingleValueTopicData` items to be rendered in the HTML page. Again these tags can appear anywhere within the HTML document. Below is an example of the syntax

```
<!--@DiffusionTag type="TopicData" name="Assets/FX/EURUSD/O" -->
```

HTTP Listener

It is also possible for Publishers to listen to all file HTTP requests by registering as a `HTTPRequestListener`. This exposes the interface

```
void handleHTTPRequest(HTTPVirtualHost virtualHost,HTTPRequest request)
```

This enables for more detailed statistics to be captured from the HTTP request

Security

Aspects of Web Server Security

Digest Authentication

Digest Authentication can be utilised to negotiate credentials with a user's web browser. It is applied to specific directories on your web site. The protection of one directory automatically applies protection to all lower directories

Copyright 2013 Push Technology

as well. The configuration file *etc/WebServer.xml* is used to add new realms and to store the users name plus the passwords.

Copyright 2013 Push Technology

Copyright 2013 Push Technology

Chapter 8

Publishers

Topics:

- *Publisher Basics*
- *Writing a Publisher*
- *Testing a Publisher*
- *Client Queues*
- *Client Validation*
- *Client Geo and WhoIs Information*
- *Client Groups*
- *Client Notifications*
- *Design Patterns*

A 'Publisher' lies at the core of the Diffusion™ infrastructure. It is a Publisher that publishes Messages to Clients who are subscribed to Topics that are provided by the Publisher. There can be one or more Publishers implemented within a Diffusion™ Server. A Publisher can provide the behaviour of one or more Topics but a Topic can only belong to one Publisher. The Publisher infrastructure is provided by Diffusion™ and the behaviour is provided by the user by writing a Publisher. A Publisher is written in Java, and deployed within the Diffusion™ Server itself. The following sections describe in more detail how to write a Publisher and describe some of the facilities available to a Publisher.

Publisher Basics

An Introduction to Publisher Architecture and Terminology.

A Publisher is a user defined object deployed within a Diffusion™ Server which provides one or more Topics on which it publishes Messages to Clients.

There may be one or more Publishers deployed with a Diffusion™ Server.

Clients connect to the Server and subscribe to Topics. Messages relate to Topics and when a Publisher publishes a message it is broadcast to all Clients that are currently subscribed to the Message Topic. A Publisher may also send Messages to individual Clients and receive Messages sent from Clients. Clients may request (fetch) Topic state, even when not subscribed.

A Publisher must be written by the user in Java (utilising the Publisher API) and deployed within the Server. This is done by extending a supplied Publisher class and implementing methods as required. You only need to implement the methods relating to the functionality that you require. See "Writing a Publisher" for more details.

Defining Publishers

How to define Publishers that start with Diffusion™.

The Diffusion™ server is able to start the Publishers defined in the *etc/Publishers.xml* file when the server starts. The XML file can contain any number of Publishers. Each Publisher must have at least a name and a class. The class must implement the Publisher by extending the `Publisher` class (See '*Creating a Publisher Class*').

```
<publishers>
 <publisher name="Publisher">
  <class>com.example.Publisher</class>
 </publisher>
 ...
</publishers>
```

The name must be unique on the server, and the class must exist on the classpath of the Diffusion™ server (See '*Classic Deployment*'). This is sufficient for the Publisher to start when Diffusion™ does. There are other options, including those that can prevent the Publisher from starting.

When the 'enabled' element is false the Publisher class will not be loaded. If the 'start' element is false then the Publisher will not be started when the server starts. If the 'topic-aliasing' element is false then '*Topic Aliases*' will not be used by Topic Messages. Below is an example with the default values for these optional settings.

```
<publishers>
 <publisher name="Publisher">
  <class>com.example.Publisher</class>
  <enabled>true</enabled>
  <start>true</start>
  <topic-aliasing>true</topic-aliasing>
 </publisher>
</publishers>
```

It is possible to define properties in the *etc/Publishers.xml* that can be accessed from the Publisher, see '*Publisher Properties*' for details.

See the '*Publisher Client Connection*' section to see how server connections can be configured in the *etc/Publishers.xml* for connecting to other Publishers.

The full configuration file options can be found in the XSD document for the *etc/Publishers.xml* or '*XML Configuration - publishers*'.

Loading Publisher Code

This describes how to load Publisher classes or code it is dependent upon.

Copyright 2013 Push Technology

When you write a Publisher class (or any other classes it will use) then you can deploy them in any folder as long as it is specified in the configuration (usr-lib in *etc/Server.xml*). Jars can also be deployed in user libraries and therefore any other software libraries that the Publisher requires may be specified in this way.

Also, when Diffusion™ starts, the `data` and `etc` folders are on the class path. The `ext` folder, and its sub-directories will be scanned for jar files and class loaded. This means that you can easily add new jars to the Diffusion™ runtime, without having to edit the startup scripts.

Caution, needs to be taken when creating backup jars in the `ext` folder as anything that ends in .jar will be class loaded.

Programmatically loading Publishers

How to load Publishers through the API

It is possible to configure and load custom Publishers via the Diffusion™ API at any point in the server's lifecycle. Similarly to loading publishers via configuration files, each Publisher must have at least a name and a class. The class must implement the Publisher by extending the `Publisher` class (See '*Creating a Publisher Class*')

```
PublisherConfig config =
 ConfigManager.getServerConfig().addPublisher("MyPublisher",
 "com.acme.foo.MyPublisher");
Publisher publisher = Publishers.loadPublisher(config);
```

The name must be unique on the server, and the class must exist on the classpath of the Diffusion™ server (See '*Classic Deployment*'). By default the `autostart` property is enabled on the PublisherConfig, so the Publisher will start once it is loaded. If this option is disabled it is possible to load a publisher and retain a reference to it, to start at a later point in time.

If the default configuration options are suitable for your requirements (as detailed within the API docs for `com.pushtechnology.diffusion.api.config.PublisherConfig`) there are several convenience methods that can be used to load a given publisher and get a reference to it without the need for construction a specific `PublisherConfig` instance.

```
// Create Publisher with classname
Publisher publisher = Publishers.createPublisher("MyPublisher",
 "com.acme.foo.MyPublisher");

// Create Publisher with Class
Publisher publisher = Publishers.createPublisher("MyPublisher",
 MyPublisher.class);
```

It is also possible to load a default Publisher instance. This will facilitate programmatic access any features exposed via the Publisher abstract class that do not require method overriding.

```
Publisher publisher = Publishers.createPublisher("MyDefaultPublisher");
```

Starting and Stopping Publishers

What happens when Publishers are started and stopped

Typically Publishers are started when the Server starts but it is possible to prevent such automatic start up and allow Publishers to be started using System Management.

Publishers may also be stopped and restarted using System Management functions and are automatically stopped and removed when the Server closes.

In order for a Publisher to function properly on being stopped and restarted from System Management it must be able to cater for the integrity of its data and Client connections. For this reason a Publisher cannot be stopped by default and must override the `isStoppable` method to enable this functionality.

Copyright 2013 Push Technology

Publisher Startup Steps

When a Publisher is started it goes through its initial processing in the order shown below:-

Table 7: Start Publisher

Add Initial Topics	Initial Topics configured for the Publisher are added.
Load Server Connections	Server Connections configured for the Publisher are loaded and validated.
`initialLoad`	The `intialLoad` notification method is called. This may be used to perform any initial processing required for the Publisher. Topics may be added here. Other aspects of the Publisher, such as Topic Loaders and Client Listeners may also be set up here. If an exception is thrown by this method the Publisher will fail to start.
Connect to Servers	A connection is made to each Server Connection .
STARTED	At this point the Publisher is considered to have started.
`publisherStarted`	The `publisherStarted` notification method is called.

Publisher Closedown Steps

When a Publisher is stopped, either during Server Closedown or by System Management it goes through the following steps:-

Table 8: Stop Publisher

`publisherStopping`	The `publisherStopping` notification method is called to allow the Publisher to perform any preliminary close processing.
Remove Topics	All Topics owned by the Publisher are removed.
Close Server Connections	Any Server Connections made by the Publisher are closed.
STOPPED	At this point the Publisher is considered to be stopped.
`0publisherStopped`	The `publisherStopped` notification method is called.
Client Events Stopped	Client Event Notifications are stopped.

Publisher Removal

A Publisher is removed after it is stopped during Server closedown but it is also possible to remove a stopped Publisher at any time using System Management. Once removed a Publisher may not be restarted again until the Server is restarted.

In either case, after removal the `publisherRemoved` notification method is called.

Publisher Topics

Topics are the Mechanism by which Publishers provide data to Clients

Each Publisher may provide one or more Topics but each Topic must be unique by name within the Server. Topics are hierarchical in nature and so Topics may be parents of Topics and a Tree of Topics can be set up. Using hierarchies allows Clients to subscribe to branches of the hierarchy rather than having to subscribe to individual Topics. Only the owner of a Topic can create new Topics below it in the hierarchy.

Copyright 2013 Push Technology

Adding Topics

In the simplest case a Publisher can name the Topics it provides within its configuration. In this case such Topics are automatically added as the Publisher is started. These Topics may be obtained from within the Publisher using the `getInitialTopicSet` method.

More typically a Publisher would add the Topics it requires itself as it starts up. A Publisher may choose to add some Topics at start up and others later. Topics can be added at any time using the Publisher's `addTopic(s)` methods. They can be added only if they are added by the owner of the parent Topic.

A Topic may be a simple Topic where all of the handling of the Topic state is provided by the Publisher. Alternatively a Topic may be created with Topic Data which handles the state of the Topic automatically.

As soon as a Topic has been added Clients can subscribe to it.

Loading Topics

Simple Topic processing involves sending all of the data that defines a Topic (the Topic Load) to a Client when they first subscribe and then subsequently sending deltas (or changes to the data). There are two mechanisms for performing the Topic load:-

- Send on Subscribe. - When the Publisher is notified of subscription it creates, populates and sends a Topic Load Message to the Client.
- Topic Loaders. - Define a Topic Loader for the Topic which is automatically called to perform the Topic Loading when a Client subscribes.

If a Topic has Topic data then the current state is automatically provided to a Client when they subscribe.

Subscribing Clients to Topics

Clients normally request subscription to a Topic and if the Topic exists then they will become subscribed to it at that point.

It is possible that a Topic does not exist at the time that a Client subscribes to it - this is called pre-emptive subscription. In this case, when a Publisher creates a new Topic it can subscribe any Clients that have pre-emptively subscribed to it by using the subscribeClients method specifying `force=false` or this could happen automatically if the parent node had automatic Topic subscription specified

A Publisher may also force all currently connected Clients to become subscribed to a Topic by calling `subscribeClients` with `force=true`.

Subscribing Clients to Topics that they were already subscribed to causes the Topic load to be performed again.

A Publisher may also cause individual Clients to be subscribed to a Topic using the Client's subscribe method or unsubscribed using the unsubscribe method.

Providing Topic State

The Publisher method `fetchForClient` must be implemented if Clients are to obtain state using the Topic fetch feature.

Handling Topics that do not Exist

A Topic is an entity that notionally has 'state' but in some circumstances a Client could request access to a Topic that does not exist. Client notifications provide a mechanism whereby this situation can be handled.

Where a Client attempts to subscribe to a Topic that does not exist then a `clientSubscriptionInvalid` notification occurs which gives the Publisher the opportunity to dynamically create the Topic (and subscribe the Client to it) if that is what is required.

Where a Client attempts to fetch the state of a Topic that does not exist then a `clientFetchInvalid` notification occurs which gives the Publisher the opportunity to return a response to the fetch request (using sendFetchReply)

Copyright 2013 Push Technology

even if the Topic does not exist. This can provide an efficient request/response mechanism without the overhead of actually creating Topics.

If a Client attempts to send a message to a Topic to which it is not subscribed, including the case where that Topic does not exist, then a `clientSendInvalid` notification occurs, allowing the Publisher to either create, subscribe the client, process the message as stand-alone data or discard the message as appropriate.

Removing Topics

A Publisher may also remove Topics at any time using its `removeTopic`(s) methods. Removing a Topic will also remove any Topics beneath it in the Topic hierarchy.

Removing a Topic causes all Clients that are subscribed to it to be unsubscribed.

Receiving and Maintaining Data

How a Publisher receives and maintains its Data

A Publisher may obtain the data it wishes to publish and transform that data in any way that is appropriate.

The Publisher will maintain the state of its own data by updating it whenever any changes are received so that as a new Client subscribes it can be sent the latest state of the data as a whole. As such changes are received they are also published as deltas to all currently subscribed Clients.

Receiving Messages from an Event Publisher

A typical way of receiving data is from an Event Publisher where messages for Publisher Topics are routed to the Publisher via its `messageFromEventPublisher` method.

If a Publisher uses Event Publishers then it should add a listener for Event Publisher connection events using `Publishers.addEventListener`. The Publisher can implement EventConnectionListener and add itself or it can use some other object. This interface allows the Publisher to be notified whenever Event Publishers connect or disconnect.

Receiving Messages from a Remote Service

Remote Service may also provided a data feed into a Pubisher. The Remote Service can publish to Topics and the updates are applied to the Topics and passed onto subscribed clients.

Publishing and Sending Messages

PublishingMessages to Clients and Sending Messages to Clients

Creating Messages

Messages can be created using the factory methods on the Publisher or on a Topic for creating Messages (called `createLoadMessage` and `createDeltaMessage`).

If within a class that does not have a direct reference to the Publisher or Topic objects then the equivalent static methods in the `Publishers` class can be used. Messages may then be populated with data using the many variants of the '`put`' method.

Publishing Messages

Messages (whether load or delta) can be sent to all Clients that are subscribed to the Message Topic using the Publisher's `publishMessage` method.

Messages can be sent to an identified group of Clients using Client Groups.

Copyright 2013 Push Technology

Exclusive Publishing

You may wish to publish a Message to all but a particular Client. For example a Message may be sent to the Publisher from a Client and the Publisher can then publish the Message to all of the other subscribed Clients.

This is done using the Publisher's `publishExclusiveMessage` method. This facility also exists on Client Groups.

Sending Messages to Individual Clients

To send a Message to an individual Client the `Client.send` method can be used.

To send a Message to a group of Clients the `ClientGroup.send` method can be used.

Publisher Notifications

Events notified to Publishers

A Publisher is notified of certain events by certain methods on it being called. These methods may be overridden by the user to perform processing at these points as required. By default these methods (other than those indicated) perform no processing. None of the methods have to be overridden and in many instances most would not need to be overridden.The notification methods are:-

Table 9: Notification methods

`initialLoad`	Called when the Publisher is first loaded. Would typically be overridden to perform any initial processing required to prepare the Publisher.
`publisherStarted`	Called after initialLoad (see startup steps).
`subscription`	Called when a Client subscribes to a Topic that the Publisher owns. References to the Topic and the Client are passed and also a flag to indicate if the Topic has already been loaded by a TopicLoader. If the Topic has not been loaded already then typically a Publisher would want to send an Initial Load message to the Client at this point. It may not be necessary to override this method if Topic Loaders are in use.
`unsubscription`	Called when a Client unsubscribes from a Topic that the Publisher owns.
`messageFromClient`	Called when a Message is received from a Client. References to the Message and the Client are passed.
`messageFromEventPublisher`	Called when a Message is received from an Event Publisher. References to the Message and the Event Publisher are passed.
`messageFromServer`	Called when a Message is received from a Server Connection. References to the Message and the Server Connection are passed.
`fetchForClient`	Called when a Client requests a 'fetch' of the Topic state for a Topic that does not have a `TopicFetchHandler`declared and does not use TopicData.
`messageNotAcknowledged`	Called when a Message which required Acknowledgement was sent by the Publisher and was not

Copyright 2013 Push Technology

	acknowledged by one or more Clients within the given timeout period.
`serverConnected`	This is called when a Server Connection is made. A reference to the Server Connection is passed.
`serverTopicStatusChanged`	This is called when a Topic subscribed to at a `ServerConnection` has it's status changed (e.g. is removed). The topic name and reference to the Server Connection is passed.
`serverDisconnected`	This is called when a ServerConnection is lost. A reference to the Server Connection is passed.
`publisherStopping`	This is called when the Publisher has been requested to stop. It gives the Publisher the opportunity to tidily perform any close processing.
`publisherStopped`	This is called after a Publisher has stopped. The Publisher may still be restarted (but only if isStoppable is true).
`publisherRemoved`	This is called when a Publisher is removed and provides the opportunity for final tidy up. The Publisher cannot be restarted after this is called.
`systemStarted`	This is called when the Diffusion™ System has completed loading and is ready to accept connections. Publishers are started before connectors, so this notification is used all Diffusion™ sub systems are loaded.

Publisher Notification Threads

In order to understand issues of concurrency when writing a Publisher it is necessary to understand in which threads the various Publisher Notifications occur.

When a Message or request is received from a Client (of any sort), Event Publisher or Server Connection then the Inbound Thread Pool is used to process it. Depending upon the number of threads in the pool this can mean that the Publisher could receive such notifications concurrently.

Message acknowledgement notifications use the Background Thread Pool.

Other notifications come from various control threads.

All of the above considerations mean that concurrency must always be taken into account in Publisher code and it must be made threadsafe as appropriate.

Client Handling

Handling Clients from within Publishers

Closing/Aborting Clients

A Publisher may close a Client at any time using the close method. This disconnects the Client which may choose to subsequently reconnect.

Alternatively a Publisher can use the abort method which sends an abort notification to the Client before disconnecting it. A Client receiving an abort notification should not reconnect.

Copyright 2013 Push Technology

Client Notifications

A Publisher may choose to receive additional Client Notifications so that it can be informed when Clients connect, disconnect etc.

Client Pings

A 'Client Ping' Message is one that may be sent to a Client which will reflect it back to the Server to measure latency. A Publisher may send a Ping Message to a Client using the `Client.ping` method and will receive a response on the `ClientListener.clientPingResponse` method within which the Message passed may be queried to establish the round trip time.

Client Message Filtering

It is possible to filter the Messages that get queued for any particular Client (see *Message Filters*)

Publisher Properties

Using properties within a Publisher

Properties for a Publisher are defined in the *etc/Publishers.xml* configuration file. As well as the standard properties a Publisher may have user defined properties. These properties may be read using convenience methods available on the Publisher (e.g. `getProperty`, `getIntegerProperty` etc).

Using Concurrent Threads

Using threads from within a Publisher

Often within a Publisher there may be a need to initiate some processing in a separate thread so that the Publisher itself is not blocked. For example, a thread could be used to poll data from some data source.

Diffusion™ provides a mechanism for easily managing concurrent processing via the threads API.

Publisher Logging

Logging from within a Publisher

Every Publisher is assigned it's own Logger which may be used within the Publisher itself for logging diagnostic messages. This Logger is obtained using the getLogger method.

The log level of the Publisher may be changed dynamically at any time using the `setLogLevel` method.

Server Connections

Connecting to other Diffusion™ Servers from within a Publisher

Publishers may act as Clients of other Publishers for the purpose of distributed processing. In this case there is a client/server relationship between two Publishers. One Publisher can be a Client of many other Publishers and a Publisher could have many Publisher Clients.

A Publisher acting as the server in such a relationship will see the Client as a normal Client. The only thing distinguishing it will be it's Client type (obtained using `Client.getClientType`).

However, for a Publisher to act as the Client of another Publisher it must make an outbound connection to the Diffusion™ Server that hosts the server Publisher. In fact, the Client Publisher would know nothing of the server Publisher, only the Server and the Topics it wishes to subscribe to, as if it were a normal Client.

Server connections can be made automatically for a Publisher by declaring them in *etc/Publishers.xml* . In this case the connections will be made automatically during Publisher startup. It is possible to configure the behaviour when such connections fail. It could cause the Publisher to fail (if it is dependent upon the server as a data source) or it could allows the Publisher to start but retry the connection periodically until it succeeds. Alternatively it could do nothing.

Copyright 2013 Push Technology

Alternatively, the Publisher can dynamically make Server Connections as, and when required. To do this it would use the `addServerConnection` method to create a `PublisherServerConnection` object, configure the connection as required and then use the connect method on the object to make the connection

Whether Server Connections are made automatically or manually the Publisher will be notified when a connection is made via the `serverConnected` method and when the connection is closed or lost via the serverDisconnected method.

The Publisher receives messages from Server Connections on the `messageFromServer` notification method.

The Publisher is notified of the change of status (e.g. removal) of any Topics it is subscribed to at a Server Connection on the `serverTopicStatusChanged` notification method.

General Utilities

General purpose utilities that may be used from within a Publisher

There are a number of general purpose utilities available which may aid in the process of writing a Publisher, for example:-

Table 10: General Publisher Utilities

Utils	A set of general purpose utilities which include file handling, property handling, date and time formatting and more.
XMLUtils	A set of utilities to aid in the processing of XML.
HTTPUtils	A set of utilities to aid in HTTP processing.

Writing a Publisher

How to approach writing a Publisher

'*Publisher Basics*' discusses the general concepts associated with Publishers. This section goes into a little detail on what actually needs to be written in Java terms. Note that this section only covers the main aspects of the Publisher API and the issued javadoc should be consulted for full details.

There are 'demo' Publishers issued with Diffusion™ which have the source provided and these act as examples of working Publishers.

In its simplest sense a Publisher is responsible for providing Topics, and publishing Messages relating to those Topics and therefore the sections on Topics and Messages should be read in conjunction with this section.

Before a Publisher is written you need to carefully consider what it needs to do and thus what methods need to be implemented. The areas that need to be considered and the methods relating to them are discussed below.

There may be many ways to approach the design of a Publisher. See '*Design Patterns*' for some possibilities.

Creating a Publisher Class

How to create a Publisher Class

A Publisher is written by extending the abstract `Publisher` class (see Publisher API) and overriding any methods that need to be implemented in order to achieve the functionality required by the Publisher. In all but the simplest of Publishers it is likely that other classes would need to be written to perform the functionality required of the Publisher.

The considerations of which methods need to be overridden are discussed further within this section.

Once the class is written and compiled it may be deployed in the Diffusion™ Server by specifying its details in *`etc/Publishers.xml`*

Publishers can also be deployed as a DAR file, sidestepping *`etc/Publishers.xml`*

Copyright 2013 Push Technology

See the section on testing for information on how to test the Publisher.

Publisher Startup

Starting a Publisher within the Diffusion™ Server

When a Publisher is first loaded by the Diffusion™ Server it may also be automatically started. If not automatically started (or if it has been manually stopped) then a Publisher can be manually started via the System Management interface. In either case the Publisher processing will go through a number of startup steps. During these steps the `initialLoad` and `publisherStarted` methods are called and these methods may be implemented by the Publisher to perform any initial processing like setting up the initial data state or adding initial Topics.

Data State

Maintaining the Data within a Publisher

A Publisher will typically be holding some data which it will update according to data update events it might receive.

The data held by the Publisher is referred to as its 'state'. To be more specific, it is the Topics provided by the Publisher that have data state. It is up to the Publisher whether the data state is managed as a whole or on Topic by Topic basis.

It is the responsibility of the Publisher to initialise its state and keep it updated as appropriate. Clients that subscribe to Topics will usually want to know the current state of the data relating to that Topic and the Publisher provides this as an 'initial load' message. Clients will then be notified of all changes to that state by the Publisher sending out 'delta' messages.

A Publisher will therefore typically have its own data model represented by classes written to support the data for the Publisher. Ways in which such a data model can be managed are discussed in '*Data Models*'.

Initial State

A Publisher's data typically needs some 'initial state' which may then be updated during the life of the Publisher. The state clearly needs to be set up before a Client requires it but exactly when this is done is up to the Publisher.

The state of the data as a whole could be set up when the Publisher starts. This could therefore be done in the `initialLoad` method where all Topics required could be set up and the data loaded as appropriate.

Alternatively, the state of the data relating to a Topic could be initialised when the Topic is added, which is not necessarily when the Publisher is started.

Another option is that the initial state is provided by a data feed as it connects (or is connected to). When input is provided by an Event Publisher then the initial state could be set up when the Event Publisher connects or the Event Publisher could send an 'initial load' Message as its first message to the Publisher. Similarly, if data is provided by a Server Connection then the initial state could be set up when the Server Connection is is notified to the Publisher or more typically an 'initial load' message would be provided by the Server.

Data Integrity

The integrity of the data is also the responsibility of the Publisher and care must be taken to ensure that all updating of data state is thread-safe. For example, it must be borne in mind that a Client could request a load of current state (e.g. by subscription) at the same time as the state is being updated (e.g. by `messageFromEventPublisher`). Note that the Topic Data feature automatically handles such data locking and in other cases Topics may be locked as and when required.

Providing Data State

If Clients are to use the 'fetch' facility to obtain the current state of Topics then it will be necessary to consider the implementation of either the `fetchForClient` method of the Publisher or a `TopicFetchHandler`.

Data Inputs

How Data can be input to a Publisher

Copyright 2013 Push Technology

For a Publisher to be able to publish data to Clients it clearly needs a source for that data. This could be obtained from some type of feed, perhaps provided by some external API or it could be from some other application communicating using Diffusion™ protocols. This is entirely up to the Publisher but Diffusion™ does offer some mechanisms.

Event Publishers

Diffusion™ provides a mechanism for feeding data to Publishers via the Event Publisher interface. The Event Publisher itself might also be written in Java using the Event Publisher API or it could be written in any other language using the Event Publisher protocol.

If a Publisher is to receive Messages from an Event Publisher then it would need to implement the messageFromEventPublisher method to process such messages. This could transform the data as required and update its own data state. Alternatively, it could specify Topic Listeners to handle the Messages on a per Topic basis.

For a Publisher to know when an Event Publisher connection is accepted by its Server or its connection is closed or lost it should add an EventConnectionListener using `Publishers.addEventListener`. The Publisher itself may implement `EventConnectionListener` or may delegate to another class. The listener will be informed of connection via the `eventConnectionAccepted` method and of disconnection via `eventConnectionClosed`. These methods pass a reference to the EventConnection so that the Publisher can determine to which Event Publisher it relates.

It should be noted that an Event Publisher may have connected before the Publisher started and therefore is not a reliable source of initial state. You will need to consider how a Publisher will establish its initial state in conjunction with an Event Publisher.

Server Connections

A Publisher may receive its data from another Publisher in a distributed environment as if it were a Client of that Publisher. In order to do this it is necessary for the Publisher to connect to the Server of the remote Publisher and subscribe to its Topics.

Connection to remote Servers can be set up automatically for a Publisher if so configured in *`etc/Publishers.xml`* . A reference to such a Server Connection can be obtained by the Publisher using the `getServerConnection` method or all connections van be obtained using `getServerConnections`.

Alternatively, a Publisher can explicitly connect to remote Servers using `addServerConnection`.

However the connection is made, the Publisher will be notified of the connection via the `serverConnected` method.

If a Server Connection is closed by the remote Server or lost the Publisher will be notified via the `serverDisconnected` method.

Once a ServerConnection is established the Publisher must subscribe to Topics on it and will then receive Topic Messages via the `messageFromServer` method. Typically, the first such method would be a 'load' message providing the initial state for the Topic. Processing of the Messages may be done within this method or alternatively the Publisher could specify Topic Listeners to handle the Messages on a per Topic basis.

See the '*Publisher Clients*' for more information.

Remote Services

A Publisher may receive input from a Remote Service.

Remote Services can publish Messages to Topics. Where such Topics have Topic Data the Topic state is automatically updated and deltas are published to subscribed Clients. Where Topics do not have Topic Data then published Messages are forwarded to subscribed Clients (i.e. it ia sssumed that the Remote Service is maintaining the data state).

Remote Services may also send messages to specific Clients and these are forwarded to the Clients automatically.

Remote Services may also send Messages to the Publisher using the `RemoteService.sendToPublisher` method. The Publisher would need to add a `RemoteControlTopicDataListener` to the Remote Control Topic Data to receive such Messages and would receive them on the `messageFromService` method.

Copyright 2013 Push Technology

Handling Client Subscriptions

How to handle Client subscriptions to Topics in a Publisher

Clients subscribe to Topics provided by Publishers and whenever this occurs the Publisher is notified via its subscription method. The Publisher may perform any processing it requires on subscription, some of the more common actions are described below.

Topic Loading

Typically, on subscription, the Publisher would want to provide the Client with the current state of the data for the Topic. It can do this by creating a new 'Topic Load' message and populating it with a representation of the state. Rather than doing this every time a Client subscribes it is generally more efficient for the Publisher to create a 'Topic Load' Message only when the state changes and send this same Message out to every Client that subscribes.

This provision of the current state is known as the 'Topic Load'. This can be done in one of the following ways:-

Topic Load in `subscription` method

If the Topic has not already been loaded by a Topic Loader (see below) then the loaded parameter of the subscription method will be false. In this case the normal action would be for the Publisher to send a 'Topic Load' Message to the Client (passed as a parameter to subscription) via its send method.

Topic Loaders

A Topic Loader is an object that implements the `TopicLoader` interface and can be used to perform any Topic Load processing that is required for one or more Topics. Topic Loaders may be declared for a `Publisher` using the `Publisher.addTopicLoader` method. This is typically done in the `initialLoad` processing and must be done before any Topics that will be loaded by the Topic Loader are added.

Using Topic Data

The use of Topic Data is recommended for Topic data management and if it is in use then Topic Loading is fairly automatic and the default `subscription` method deals with it.

Hierarchic Subscription

When a Client subscribes to a Topic the Publisher may choose to subscribe the Client to other Topics or to subordinate Topics. This can be done using the `Client.subscribe` methods.

A Client itself can request subscription to a hierarchy of Topics using Topic Selectors but this is an alternative method of handling hierarchies.

Publishing Messages

How to Publish Messages

Publishing a Message means sending it to all Clients subscribed to a Topic. The message itself nominates the Topic to which it relates.

The most common reason for publishing a Message would be to send an update (or delta) to the Topic's data to all subscribed clients but messages can be published for any reason. In the case of an update then this would be done in the method that became aware of the update (e.g. `messageFromEventPublisher`, `messageFromServer` etc).

A message for publishing may be created and populated by the Publisher and then published using publishing methods on the Topic or the Publisher itself.

Exclusive Messages

If the Publisher wishes to publish a Message to all Clients subscribed to a Topic except one single client then it can use the publishExclusiveMessage method. This may be appropriate if the Message being published is a result of receiving a Message from a Client which you do not want to send back to that Client.

Copyright 2013 Push Technology

Message Priority

The priority at which a message is to be sent can be altered from the normal priority. For example, an urgent message can be sent with high priority causing it to go to the front of the Client's queue.

Message Acknowledgement

If acknowledgement of a message is required then it can be set as an 'Acknowledged Message'. If any Clients do not respond to a acknowledged Message within a specified timeout then the Publisher will be notified on its `messageNotAcknowledged` method.

Topic Locking

The Publisher must consider the issue of locking the Topic whilst it's state is changed and delta messages published. When TopicData is in use then all locking of the Topic state is handled automatically. However, when the state of the Topic is maintained in some other manner (for example as a Topic attachment) then it is the responsibility of the Publisher application to handle locking as described below.

By default all Topics have locking enabled which allows the Publisher to lock and unlock the Topic as required. When a Client subscribes to a Topic the subscription method of the Publisher would normally send the current state of the Topic to the Client. With locking enabled the Topic will be locked for the duration of the subscription thus preventing other threads from acquiring the Topic lock.

Threads that update the Topic state and publish messages should also lock the Topic for the duration of the update and publish. If this technique is not employed then it is possible that a delta message could be sent to a client before the subscription method has the opportunity to send a Topic Load message. This could cause a failure at the Client if Topic aliasing is in use as aliasing relies upon the Topic load message reaching the Client first. Even if Topic aliasing were not in use the Client application would have to be prepared for a delta arriving before the Topic load.

The following code example shows how a method responsible for updating the Topic state and publishing a delta should handle Topic locking:-

```
void updateMyData(String newValue) throws APIException {
    theTopic.lock();
    try {
        MyData data = (MyData)theTopic.attachment();
        data.setValue(newValue);
        TopicMessage message = theTopic.createDeltaMessage();
        message.put(newValue);
        publishMessage(message);
    }
    finally {
        theTopic.unlock();
    }
}
```

Using the try/finally technique ensures that whatever happens the code that locks the Topic will also unlock thus preventing a lock being left on indefinitely. As mentioned earlier the Topic is automatically locked during subscription so the above code prevents an update from occurring whilst a subscription is in progress.

Handling Clients

Interacting with Clients from within a Publisher

A Publisher is notified when a Client subscribes to one of its Topics via the `subscription` method and when the Client unsubscribes the `unsubscription` method is called.

A Publisher may receive Message from Clients and send Messages to Clients (see below).

Copyright 2013 Push Technology

A Client may request the state of any Topic(s) at any time even if not subscribed to it. This is referred to as as 'fetch' request. Such a request may routed to the Publisher's `fetchForClient` method if a Topic has no `TopicFetchHandler` declared and does not use TopicData.

Other than the above, a Publisher is not normally notified of any other Client activity. However a Publisher can choose to receive Client Notifications using the `Publishers.addEventListener` method. Using Client notifications a Publisher could even handle a fetch request for a Topic that does not exist and return a reply (using `Client.sendFetchReply`) without the overhead of acually creating a Topic.

A Publisher can also choose to close or abort Clients.

Sending and Receiving Client Messages

In addition to publishing Messages to all Clients subscribed to a Topic it is also possible to send a message to a single Client only using the `Client.send` method.

A Client can also send Messages to the Publisher and these would be received on the `messageFromClient` method which should handle them accordingly. This method need only be implemented if Messages are expected from Clients. Alternatively the Publisher could specify Topic Listeners to handle the Messages on a per Topic basis.

The 'Client Groups' facility allows Messages to be sent to all Clients in a group. There is also an 'exclusive Message' facility which caters for sending to all but one Client in a group.

Publisher Closedown

What happens when a Publisher is closed

A Publisher is stopped and removed when the Diffusion™ Server closes but can also be stopped and restarted, or stopped and removed via the System Management interface.

However a Publisher is stopped it will always go through a set of closedown steps, during which the `publisherStopping` and `publisherStopped` methods will be called. A Publisher may implement these methods if required to perform any special processing such as tidying up resources used.

Publisher Removal

When a Publisher is finally removed (either during Server closedown or via System Management) then it cannot be restarted again within the same Server instance. After removal the publisherRemoved method is called and this gives the Publisher the opportunity to perform any final tidy up processing.

Stopping and Restarting via System Management

By default it is not possible to stop and restart a Publisher using the System Management functions because in order for this to work the Publisher needs to cater for the integrity of its state when this happens. As Topics are also removed during stopping then the Publisher would also need to be able to restore these Topics if it were restarted.

If a Publisher does want to cater for stop and restart via System Management then it must override the `isStoppable` method to return true. The Publisher code must then be able to recover Topics and data state on restart.

Testing a Publisher

How to test a Publisher

Having written a Publisher and configured it with a Diffusion™ Server instance then there are various ways in which it can be tested.

The easiest way to perform some initial tests is to start it and try it out using some of the supplied testing tools.

For example, you could start one or more instances of the Client Test Tool, connect each to the test Server and subscribe to the Publisher's Topic(s). You should see the Initial Topic Load data displayed and any messages sent as

Copyright 2013 Push Technology

deltas would also be displayed in each Client. This tool may also be used to send messages to the Publisher from the Client.

You can simulate data being sent from an Event Publisher using the Event Publisher Test Tool.

Ultimately such tests are limited and you may wish to develop Java tests which simulate Clients using the Java External Client API (or the Windows External Client API).

Ideally you would test as soon as possible with the actual Clients that are going to be used. So, for example, you may wish to develop browser clients using HTTP, Flash or Silverlight.

It may help to diagnose problems with the Publisher if it has diagnostic logging encoded within it. Such logging could be provided only at 'fine' level and this logging level used only during testing.

Client Queues

How Messages sent to Clients are queued and how such queues can be manipulated by Publishers

Diffusion™ maintains an out-bound queue of Messages for each Client. Whenever a Message is published to a Topic then it will be placed in the queue of each Client subscribed to that Topic as will any message sent explicitly to the Client. Messages are then sent to the Client strictly in the order that they are enqueued.

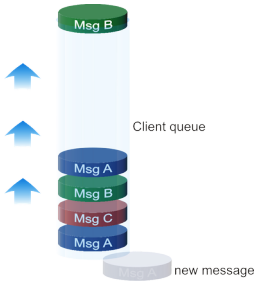


Figure 20: The Message Queue

A Publisher is able to enquire upon the details of a particular Client's queue and even to change some aspects of the queue's behaviour.

Queue Enquiries

How the Publisher can access details of Client Queues

A Publisher can enquire upon the following information about a particular Client's queue via the Client interface:-

- The Current Queue Size
- The Maximum Queue Size (The limit the queue can reach before the Client is automatically disconnected.)
- The Largest Queue Size (The largest size the Client' queue has been since the Client connected.)

Maximum Queue Depths

Limits to the extent of a Client Queue

In order to limit the backlog of Messages queued for a Client that is not consuming them quickly enough it is possible to indicate a maximum queue depth for Clients.

This size must be chosen carefully as a large queue size may lead to excessive memory usage and vulnerability to 'Denial of Service' attacks, whilst a small queue size can lead to slow Clients being disconnected too frequently.

The maximum queue depth for Clients can be configured for a Client Connector in *etc/Connectors.xml* . A default value may also be configured in *etc/Server.xml* for Connectors that do not explicitly specify a value.

By specifying a maximum queue depth of 0 you can indicate that queues are to be 'unbounded' meaning that there will be no limit on the size that they can grow to.

Copyright 2013 Push Technology

These values may be changed dynamically at run time using System Management but they will only take effect for new Clients.

Queue Notification Thresholds

Notification of Client Queue Threshold breaches

A Publisher may wish to know when a Client queue has reached a particular size so that it could act upon the event in some way.

For example, the Publisher may want to notify the Client so that it can take some action (like suspending processing). As there would be little point in queueing a message to tell the Client that their queue is becoming full then this would most probably be done using a high priority message which would go to the front of the queue.

To this end, an "upper notification threshold' can be set for a Client's queue. This is expressed as a percentage of the maximum queue depth at which a notification will be sent to any 'Client Listeners' that are declared. A 'Client listener' is any object that implements the `ClientListener` interface and such a listener can be added from within a Publisher using the `Publishers.addEventListener` method. Listeners are notified of the limit breach via the `clientQueueThresholdReached` method.

In addition a 'lower notification threshold' may be specified. The lower threshold is a percentage of the maximum queue depth at which a notification should occur when a message is removed from the queue causing the queue size to reach the threshold if (and only if) the upper threshold has been breached.

When the `clientQueueThresholdReached` method is called on the Client Listener it indicates whether it was the upper or lower threshold that was reached.

The thresholds to use for Clients may be configured for a Connector in connectors.properties. If not specified then the default thresholds specified in *`etc/Server.xml`* are used.

The thresholds on a Client Connector may be changed dynamically at run time using System Management but the new values will only take effect for new Clients.

Thresholds can also be set or changed from within the Publisher for a connected Client using the `Client.setQueueNotificationThresholds` method.

Note that this feature can only be used for 'bounded' queues. If no maximum queue depth has been specified then queue thresholds will be ignored.

Tidy on Unsubscribe

Facility to clear out Queued Messages for a Topi when unsubscribed

Once a Message is queued for a client then it's delivery is guaranteed. This means that a Client can unsubscribe from a Topic but still receive Messages queued for it on it on that Topic. This is generally what is required as the Messages were sent whilst the Client was subscribed.

However, it may be decided that once the Client has unsubscribed from a Topic then it no longer has any interest in any Messages for that Topic and such Messages should be removed from the queue. To achieve this there is an option on a Topic (using the `setTidyOnUnsubscribe` method) to indicate that Messages for the Topic should be removed from Client queues when the Client unsubscribes from that Topic.

Filtering Queued Messages

Filtering the Messages that get Queued for a Client

It is possible to filter the Messages that get queued for a particular Client. See '*Message Filters*' for more details.

Client Validation

How to use Connection and Subscription Blacklists/Whitelists

Copyright 2013 Push Technology

Diffusion™ provides the facility to validate Client connections to determine whether they are allowed using whitelists (allowed Clients) and blacklists (barred Clients). This can be made to occur automatically or can be controlled programmatically from within Publishers.

Validation can occur at the following levels:-

- On Connection (When a Client connects to Diffusion™ it is possible to validate whether the Client is allowed to connect (regardless of Topic being requested).)
- On Subscription (When a Client subscribes to a Topic it is possible to validate whether the Client is allowed to subscribe to the Topic.)

You can allow/disallow clients explicitly using their IP address (or host name) or by geographical criteria determined by Client Resolution (e.g. you could bar all clients from a particular country). You can also perform 'fuzzy' matching on address or geographical values (e.g. you could bar all Clients with a host name that starts with a specified string).

The specification of the Clients that are to be allowed/disallowed is known as a 'Client Validation Policy'. There can be one 'Client Connection Validation' Policy for a Diffusion™ Server and there can also be a 'Client Subscription Validation Policy' specified for each Publisher.

The Policies are maintained as files but may be updated programmatically from within Publisher code. So it is possible to dynamically update a Policy during Publisher processing.

The format of the XML policy files is defined by the issued XSDs and is also defined in the *configuration* section.

Client Validation Policy Types

A Validation Policy may be a Blacklist or a Whitelist

A Client Validation Policy may be a 'whitelist' or a 'blacklist'.

With a whitelist only those Clients who match the criteria specified within the Policy are allowed to connect or subscribe, whereas with a blacklist any Clients who match the criteria specified within the Policy will be prevented from connecting or subscribing.

Automatic or Manual Policies

Automtic or Manaul Validation Policies

If a Client Validation Policy is declared as being 'automatic' then it will be applied automatically by Diffusion™ and Clients that do not pass the Client Validation will automatically be disconnected (for a Connection Policy) or unsubscribed (for a Subscription Policy).

A Policy may also be defined as 'Manual' in which case Diffusion™ will not automatically apply the Policy. In this case the Policy may be manually applied using the Publisher API and the Publisher can decide what action to take, not necessarily disconnect or unsubscribe.

Client Validation Criteria

What criteria can be checked in a Validation Policy

The criteria by which a Client is validated as being allowed or disallowed is specified in terms of selection values.

The value types are as follows:-

Table 11: Validation criteria

address	The Client's IP address
host name	The Client's Host Name
resolved name	The Client's Resolved Name
country	The Client's Country Code
language	The Client's Language Code

Copyright 2013 Push Technology

The values (other than address) are as resolved by the Diffusion™ WhoIs Service therefore the WhoIs Service must be enabled if used. It should also be noted that when resolved values are used then the comparison cannot be made until the Client is resolved. This can mean that a Client that has not yet been resolved may be allowed to connect but is then disconnected when it is resolved and found to be disallowed by one of the resolved Policy criteria.

For each type the Policy may specify any number of values to check against. Each value may be an absolute value (e.g. an IP address) or a 'Regular Expression' Pattern which will be evaluated against the actual Client value.

When a Client connects or subscribes then each of its actual values are checked against the policy lists for a match.

A blacklist policy will disallow any clients that have values matching a value within any value type specified within the policy. Value types may therefore be mixed. For example you could specify a country code and a list of excluded IP addresses which would cause all clients from the specified countries to be excluded plus those with the specified addresses.

A whitelist policy would only allow clients that match at least one value from each of the specified value types. For this reason it does not usually make sense for more than one value type to be specified in a whitelist.

For example, consider the following value lists:-

```
address=194.124.6.78 & 195.233.3.4
hostName=XYZ*
```

If this were a blacklist then it would disallow the IP addresses specified plus any Client that has a resolved host name starting with 'XYZ'. However, if this were a whitelist it would only allow a Client if it had one of the specified IP addresses AND a host name starting with 'XYZ'.

Client Connection Validation Policy

A Validation Policy for checking when Clients connect

The Client Connection Validation Policy applies to Clients connecting to a Diffusion™ Server. Each connector may have its own policy file, which is specified in *etc/Connectors.xml* .

If an automatic Policy has only address information (i.e. no criteria that require Client resolution) then any Client disallowed by the Policy will not be allowed to connect. If the Policy does have criteria that require Client resolution and the Client connecting has not been resolved then it will be allowed to connect but may be disconnected as soon as the Client is resolved.

Client Subscription Validation Policy

A Validation Policy for checking when Clients subscribe to Topics

A Client Subscription Validation Policy applies to Clients subscribing to Topics belonging to the Publisher that the Policy is specified for. Each Publisher may have its own policy file, which is specified in *etc/Publishers.xml*

If an automatic Policy has only address information (i.e. no criteria that require Client resolution) then any Client disallowed by the Policy will not be allowed to subscribe to any Topic indicated by the Policy. If the Policy does have criteria that require Client resolution and the Client subscribing has not been resolved then it will be allowed to subscribe but may be unsubscribed as soon as the Client is resolved.

Using Policies Programmatically

How to use Validation Policies Programatically

Both Connection and Subscription Validation policies may be defined and changed at runtime using the configuration interface. Even if the policies were loaded from XML they can be changed at runtime, even after the server has started.

Connection Validation Policies may added to Connectors or existing Policies defined for Connectors can be amended.

Subscription Validation Policies can be defined for Publishers or exsiting Policies can be amended.

Copyright 2013 Push Technology

Client Geo and WhoIs Information

When a Client connects to Diffusion™ then information about that Client's geographic location is looked up and the information is made available to Publishers.

When a Client first connects to a Diffusion™ Server then its remote IP address is immediately available (via the `Client.getRemoteAddress` method) as well as other details obtained from the embedded GeoIp database. Further host and geographic details about the Client are then obtained using the Diffusion™ "`WhoIs` Service".

GeoIp Information

Diffusion ships with a GeoIP database from *MaxMind*. This provides information about Locale and geographic co-ordinates. The Java API includes utilities (`GeoIPUtils`) to make use of this database.

This is a public domain database and is free to use. It is possible to purchase the more accurate database from MaxMind and change the configuration in the *`etc/Server.xml`* properties to use the new database.

The database can be disabled but its use is mandatory if you are going to use client *connection or subscription validation policies*.

WhoIs

The inbuilt WhoIs service can provide additional information about clients, however the lookup of the WhoIs information may take some time, especially if it is not already cached. This means that notification of the connection and further processing of the Client cannot wait for this information to become available. For this reason the resolution of the Client's WhoIs details is notified to Client Listeners separately from Client connection on the `clientResolved` method.

When a Client is first connected it is likely that the WhoIs details of the Client are not available. This may be checked using the `Client.isResolved` method. When the details become available they may be obtained from the Client using the `getWhoIsDetails` method which returns an object containing the following information:-

Table 12: WhoIs

Address	The Client's IP Address - this will be the same as that obtained using `Client.getRemoteAddress`.
Host	The resolved host name of the Client. If the host name could not be resolved then the address will be returned.
Resolved Name	The fully resolved name of the Client. Exactly what this means would depend upon the WhoIs Provider in use. If a fully resolved name could not be obtained then the host name value will be returned.
Locale	Returns the result of a geographic lookup of the IP address indicating where the address was allocated. The country of the Locale is set to the international two-letter code for the country where the Internet address was allocated (e.g. NZ for New Zealand). If the Internet address could not be found in the database, the country and language of the returned Locale are set to empty Strings. Three country values can be returned that do not exist within the international standard (ISO 3166). These are EU (for a non-specific European address), AP (for a non-specific Asia-Pacific address) and ** (an Internet address reserved for private use, for example on a corporate network not available from the public Internet). The language of the returned Locale is set to the international two-letter code for the official language of the country where the Internet address was allocated. Where a country has more than one official language, the language is set to that which has the majority of native speakers. For example, the language for Canada is set to English (en) rather than French (fr). Non specific addresses (EU and AP), private

Copyright 2013 Push Technology

	Internet addresses (**), and addresses not found within the database, all return an empty string for language.
WhoIsData	This is data extracted from an enquiry upon a 'WhoIs' data provider.
Local	Indicates whether the Client address is a local address, in which case no Locale or WhoIsData will be available.
Loopback	Indicates whether the Client address is a loopback address in which case no Locale or WhoIsData will be available.

The Diffusion™ WhoIs Service

The service that performs resolution of Client location details

The Diffusion™ WhoIs Service runs as a background task in the Diffusion™ Server. It looks up client details and caches them in case the same Client reconnects later.

The behaviour of the WhoIs Service is configured in *etc/Server.xml*. This allows the following to be specified:-

Table 13: WhoIs service

The WhoIs Provider	This specifies a class to use for WhoIs lookups. A default WhoIs Provider is provided with Diffusion™.
Number of Threads	The number of background threads that will process `WhoIs` resolver requests. More threads will improve the WhoIs performance. Setting this to 0 disables WhoIs.
Who Is Host/Port	These details provide the location of an internet based WhoIs lookup server that adheres to the RFC3912 WhoIs protocol. This will be used by the Default WhoIs Provider. This defaults to using the RIPE database.
Cache Details	Specifying the maximum size of the cache of details and how long cache entries should be retained before being deleted. If you envisage large numbers of different Clients connecting over time then it is important to consider the automatic cache tidying options on the service.

The WhoIs service can be disabled both by setting the number of threads to zero and removing the whois configuration element.

WhoIs Providers

The Diffusion™ WhoIs Provider is a class which implements the `WhoIsProvider` interface of the `WhoIs` API. This is used by the `WhoIs` Service to lookup WhoIs details for connected clients.

Default Provider

A default `WhoIsProvider` (`WhoIsDefaultProvider`) is provided with Diffusion™.

A connection is made to the WhoIs server specified in *etc/Server.xml* and returned details are parsed and used to update the supplied details. Child details objects will be added for any separate WhoIs records found and the type of such objects would be the key of the first WhoIs record entry (e.g. "person"). Where duplicate field names occur then all but the first will be suffixed by "_n" where n is a number distinguishing the entries.

The 'netname' entry will be used as the resolved name if present.

Custom Provider

Copyright 2013 Push Technology

If the behaviour of the issued default WhoIs Provider is not exactly what is required then users may write their own WhoIs Provider which must implement the `WhoIsProvider` interface. The name of the user written class may then be specified in *`etc/Server.xml`* and must be deployed on the Diffusion™ Server's classpath.

Client Groups

Clients that are connected may be managed in 'Client Groups' allowing Messages to be sent to groups of Clients

Client Groups are a convenient way of managing groups of Clients with common attributes.

A Publisher can create a Client Group and add Clients to it. Messages may then be sent to the group of Clients rather than to individual Clients.

When a Client disconnects it is automatically removed from all Client Groups of which it is a member.

A Client Group belongs to the Publisher that created it and can only be used from within that Publisher. Group names are unique within the Publisher only.

Client Groups are a feature of the Java API via the `Publisher` interface.

Creating a Client Group

A Publisher can create a new Client Group at any time using `createClientGroup`. When creating the group it is given a name which must be unique within the Publisher. When a Client Group is created a reference to a `ClientGroup` object is returned.

Adding Clients to a Group

Clients may be added to a Client Group using the `addClient` method on a `ClientGroup` object.

Sending Messages to Clients in a Group

A message may be sent to all Clients in a Client Group using the `ClientGroup.send` method. To send to all but a specified Client use `publishExclusiveMessage`.

Removing Clients from a Group

To remove Clients from a Client Group, use the `ClientGroup.removeClient` method.

Removing a Client Group

To delete / remove a Client Group that is no longer required, use the `Publisher.removeClientGroup` method.

Other Methods

All Clients within a Client Group may be listed using `ClientGroup.getClients`.

You can check whether a particular Client is already a member of a Client Group using `ClientGroup.containsClient`.

To enquire upon which Client Groups a particular Client belongs to you can use `Publisher.getClientGroupMembership`.

To get a reference to a named Client Group from a Publisher use `getClientGroup`.

Temporary Client Groups

Normally Client Groups have Publisher Scope, but it is possible to create temporary groups using the `union` and `intersect` methods. These methods allow for the creation of Client Groups which are not managed by the Client Group Manager. These temporary groups allow for the sending of messages which contain clients from different groups.

Copyright 2013 Push Technology

Examples

The "Group Chat" demo Publisher contains examples of the use of Client Groups.

Client Notifications

How a Publisher can receive notifications relating to Clients.

A Publisher may opt to receive certain notifications regarding Clients. It does this by adding a `ClientListener` which can be the Publisher itself or any other object that implements the `ClientListener` interface.

A listener is added using the `Publishers.addEventListener` method.

All notifications are passed a reference to the Client in question which may be interrogated for further information as required.

Notifications received on the `ClientListener` interface are as follows:-

Table 14: Client Listener Notifications

`clientConnected`	This is called whenever a new Client connects. It is not necessarily a Client that is subscribing to one of the Publisher's Topics.
`clientResolved`	This is called when a newly connected Client is resolved. A Client's full geographical information is not necessarily available as soon as a Client connects and so this method is called separately after the Client has been resolved.
`clientSubscriptionInvalid`	This is called whenever a Client attempts to subscribe to a Topic that does not exist. This might be because the Topic is not yet available and this gives a Publisher the opportunity to create the Topic and subscribe the Client to it.
`clientFetchInvalid`	This is called whenever a Client attempts to fetch a Topic that does not exist. This gives the Publisher the opportunity to respond to fetch request on a non existent Topic. A Publisher may even reply to such a request without the needs for creating a Topic using the `sendFetchReply` method.
`clientSendInvalid`	This is called whenever a Client attempts to send a message to a Topic that does not exist, or to which the client is not subscribed. This enables a client to send a message to a Topic and for that topic to be created and subscribed to on demand, or send data when a response is never expected.
`clientQueueThresholdReached`	This is called whenever a Client's queue breaches an upper queue notification threshold or returns to a lower queue notification threshold. Parameters indicate which threshold has been reached and the threshold value.
`clientCredentials`	This is called whenever a Client supplies new credentials after connection. It is called after the authorisation handler (if there is one) has validated the credentials.
`clientClosed`	This is called whenever a Client disconnects. The reason for disconnection may be obtained using `theClient.getCloseReason` method.

Adding a ClientListener

How to add a ClientListener

So a Publisher could add itself as a listener for Client notifications as follows:-

Copyright 2013 Push Technology

```
public class MyPublisher extends Publisher implements ClientListener {

    protected void initialLoad() throws APIException {
        Publishers.addEventListener(this);
    }
```

Using DefaultClientListener

How to use the Default Client Listener to avoid implementing all methods.

The Publisher must implement all of the `ClientListener` methods.

For convenience, an abstract `DefaultClientListener` class is provided which has empty implementations of all methods. This can be extended to produce a Class which only implements the methods you are interested in. Alternatively an anonymous class can be used within the Publisher as follows:-

```
    protected void initialLoad() throws APIException {
        Publishers.addEventListener(
            new DefaultClientListener() {
                public void clientConnected(Client client) {
                    LOG.info("Client {} connected",client);
                }

                public void clientClosed(Client client) {
                    LOG.info("Client {} closed",client);
                }
            });
    }
```

Design Patterns

Hints and tips on how to design a Publisher.

There are many ways in which to approach designing and writing a Publisher and this may be significantly different from application to appplication.

This section provides some hints and tips and common patterns that may be employed when writing Publishers.

Data Models

In which ways can Data be represented

Almost all Publishers need to manage data in some way. It is necessary to design the data model that will hold the data within the Publisher. This needs to take into consideration the format of the data that will come into the Publisher and how it will be transformed into the Publisher data. More importantly it needs to consider the format of the data that will be sent on each Topic to clients in the form of initial load and then delta Messages.

When data is fed to a Publisher via an Event Publisher or Server Connection then it is likely that the data will already have been transformed into a suitable format for the Publisher by the sending application, but further transformation may still be required before the data is in a suitable format for the Publisher's Topics. Data from other sources will need to be transformed into formats suitable for the Topics. These data transformations represent a large part of what a Publisher does.

Topic Data

A useful pattern for handling data within a Publisher is by associating the data with the Topics themselves.

Topics have a feature called 'Topic Data' which greatly simplifies the handling of data associated with a Topic.
A number of different data types are catered for as well as the ability to write a custom data handler. Topic Data

Copyright 2013 Push Technology

provides mechanisms for data locking, data state management and Message cacheing. It also provides mechanisms for automatically generating deltas by comparing incoming data updates with the current data state.

The Topic class also has the ability to attach an object of any type. This mechanism may also be used to associate some type of data handling object with the Topic when the use of Topic Data is not appropriate.

Hierarchic Topics

Topics may be organised hierarchically

Within Diffusion™ Topics are organised as a hierarchy known as the 'Topic Tree'. This provides for a high level of control over the organisation of Topics and how they are handled. For example, it is possible for Clients to subscribe to branches of the tree instead of all of the individual Topics

Of course, a Publisher may only be handling a very small number of Topics or it may be inappropriate to organise hierarchically, in which case a flat structure could be employed.

Flat Topic Organisation

In some cases a flat Topic structure may be appropriate. Reasons for this may be as follows:-

- There are very few Topics.
- The Topics have no logical hierarchical organisation.
- Topic name lengths are to be kept to a minimum (although the use of Topic aliasing would mean there is little reason to do this).

In this case Topics are added individually within the Publisher.

Hierarchic Topic Organisation

A hierarchy of Topics may be employed when there is a need for more fllexibility of Topic control. Where data is logically hierarchical in nature anyway then it can be very advantageous to arrange the Topics to match the data hierarchy for the following reasons:-

- Fine grain data subscription. (Clients only need to subscribe to the individual Topics they want data for, thus minimising the data that needs to be sent to the Client. Also, when data changes, only the data for the Topic needs to be sent and not all data.)
- Subscription Control. (Clients can subscribe to branches of the Topic tree without subscribing to individual Topics.)

See the section on the Topic Hierarchy for more details of how to use Hierarchical Topics.

Note that when longer hierarchic Topic names are in use then the effect on Messages sizes can be considerably reduced by using the Topic aliasing feature.

Copyright 2013 Push Technology

Copyright 2013 Push Technology

Chapter 9

Clients

Topics:

- *Connection*
- *Topics*
- *Messages*
- *Client Types*
- *Client Close Reason*
- *Cross-domain*
- *Browsers supported*
- *Protocols Supported*

In Diffusion™ terms a 'Client' is any application that connects to a Diffusion™ Server using a Diffusion™ Client connection protocol.

Connection

How Clients connect to Diffusion™

A Client must connect to a Diffusion™ Server in order to communicate with Diffusion™.

Connections may be HTTP or via a TCP/IP socket connection depending upon Client Type.

A TCP Client must specify the host and port to connect to. The port will correspond to the port that a connector of the type corresponding to the Client Type is listening on at a Diffusion™ Server on the specified host machine.

If the server is configured to support Client reconnection and the Client API also supports it then a Client disconnected due to IO error can attempt to reconnect without losing Topic state or any Messages queued for the client whilst disconnected.

API specifications will give more detail on the specifics of Client connection.

Topics

How Clients subscribe to Topics

A Client may subscribe to Topics and will receive Messages on all Topics to which it subscribes.

Topics for subscription to may optionally be supplied on connection and further Topic subscriptions may be requested at any time during the connection.

A Client may subscribe to any number of topics.

Topics may be specified either by providing explicit Topic names or by using Topic Selectors.

A Client may also 'fetch' the current state of a Topic without being subscribed to it. This effectively provides an asynchronous request/response mechanism. Headers sent with the fetch request are reflected back with the reply allowing for correlation of requests and responses. It is even possible to request a fetch for a Topic that does not exist as long as the Publisher handles it, thus providing request/response without Topics.

Messages

How Clients receive Messages

Typically, after subscription a Client will be sent a Topic Load Message containing the current Topic state and from then on, until unsubscription or disconnection, will receive updates to the Topic in the form of Delta Messages.

Clients may also receive messages sent directly to them from Publishers and may also send Messages to Publishers (via their Topics). The Client may indicate that Messages sent to the Publisher 'require acknowledgement'.

Clients may 'ping' the Server with a Ping Message which causes the Server to send a response from which the latency of the connection can be established.

Client Types

There are many different types of Client that can connect to Diffusion™

The exact functionality available to a Client application may depend upon the Client type and thus the API used (if any).

There are many types of Client as follows:

Copyright 2013 Push Technology

Table 15: Client Types

External	This is a broadly generic term for any client that connects to Diffusion™ using DPT. This could be an application using the Java Client API or the .Net Client API. It could also be any other application that communicates directly using DPT. Full duplex connections over HTTP are catered for by the External Client APIs.
Javascript	This is a Client that connects from a Web Browser using the Diffusion™ Javascript Client APIs which allows Javascript to encapsulate a plug-in client (Flash or Silverlight) or use a generic transports like IFrame, Callback Polling HTTP or Websockets.
Flash	This is a Client that connects from a Web Browser using the Diffusion™ ActionScript API which enables Web Applications to exploit the Adobe Flash Platform.
Silverlight	This is a Client that connects from a Web Browser using the Diffusion™ Silverlight API which enables Web Applications to exploit the Microsoft Silverlight framework.
Web Socket	This is a Client that connects from a Web Browser that supports Web Sockets or our Java client implementation.
Publisher	A Diffusion™ Publisher may be a Client of another Diffusion™ Publisher. It is this facility that provides Distribution within Diffusion™.
Android	This is a Client that connects from an Android phone using the Android Client library, which uses the Diffusion™ Android phone API.
iOS	This is a Client that connects from an iOS device using the Objective C library, which uses the Diffusion™ iOS API.
J2ME	This is a Client that connects from a J2ME enabled phone using the provided jar, which uses the Diffusion™ J2ME API.

External Clients

An External Client is an application that connects to Diffusion™ using DPT.

An External Client may use the Java Client API or the .NET Client API or could be written in any language that can read and/write over TCP sockets.

API External Clients may connect using TCP or HTTP or other Diffusion protocols. The transports provided vary by client but the Java client provides all transports except the HTTP Comet-based transport that is only supported in Flash.

Please refer to the *Protocols Supported* section for information on which clients support which protocols.

Secure Connections

External Clients using the Client APIs can also connect over a secure (SSL) connection. Again available secure connections vary between clients. The .NET external client does not provide HTTPS, it does allow secure DPT connections to be made. If you need an external client that supports HTTPS you can use the Java external client.

Copyright 2013 Push Technology

Proxy Connections

External Clients using the Client APIs can connect via an HTTP Proxy.

Javascript Clients

Javascript Clients use a javascript library called diffusion.js.

The transport that they use to connect to the Diffusion™ Server has been encapsulated in the DiffusionClient singleton To connect to a Diffusion™ Server a connection object needs to be passed into the connect method of DiffusionClient. The full list of parameters are provided in the jsdoc

Silverlight Clients

A Silverlight Client uses the PushTechnology.Transports.dll assembly (found in the 'lib' directory of a Diffusion™ install).

API Silverlight clients may connect using DPT or HTTP; they can also be configured to cascade. This means that they will try the socket connection first, and if that fails or is blocked, an HTTP connection will be attempted.

This client can also connect over a secure (SSL) connection using HTTPS

Port Selection

When configuring Diffusion™, it is worth thinking about which port to use with Silverlight policy file. DPT uses port 943, on the other hand any port can be used with HTTP, therefore you must bear in mind the security implications.

Security Implications

As of Silverlight 2 onwards, the security rules have changed. Silverlight will try and get a policy file from port 943, if it can't get a policy file from this port, the connection will be refused. Diffusion™ understands these requests so it is important that you have supplied the correct policy file to the Silverlight connector and the Policy File connector. An example of a policy file is included in the etc directory, although this is configured to be very open.

Flash Clients

A Flash Client uses DiffusionTranport.swc (found in the lib directory of a Diffusion™ install).

Flash Clients may connect using DPT, HTTP or HTTPC. They can also be configured to cascade. This means that they will try the socket connection first, and if that fails or is blocked, then try a HTTP connection.

This client can also connect over a secure (SSL) connection using HTTPCS or HTTPS

Port Selection

When configuring Diffusion™, it is worth thinking about what port to use for the Flash socket. Any port can be used, but you must bear in mind the security implications. Clients have had good success having the port set to 443, as this port is open in most firewall situations.

Security Implications

As of Flash Player version 9, the security rules have changed. Flash will try and get a policy file from port 843, if it can't get a policy file from port 843, then it will use the port configured for the connection to get a policy file. Diffusion™ understands these requests so it is important that you have supplied the correct policy file to the Flash connector and the Policy File connector. An example of a policy file is included in the etc directory, although this is configured to be very open.

In order to use HTTP or HTTPC a cross-domain policy has to be configured. Please refer to the Cross-domain section for more information.

Websocket Clients

A Client can connect over the Websocket Protocol

Web Socket Clients

Web Socket Clients can be used via the Javascript API, .Net or the Java API.

Copyright 2013 Push Technology

Web Socket clients use the TLS / SSL protocol if given the wss: schema.

Publisher Clients

A 'Publisher Client' refers to a Publisher which is connected to a Diffusion™ Server as if it were a Client for the purpose of Distribution.

When a Publisher wants to connect to a Diffusion™ Server it must specify the host and port of a Client Connector at the Server it wants to connect to and then subscribe to Topics on that connection (i.e. behave exactly like any other Client).

A Publisher may be caused to make such Server connections either by configuring them in *etc/Publishers.xml* or programatically using `Publisher.addServerConnection`.

Android Clients

This is an Android client which adheres to the Diffusion™ Client API. It has been written in Java using the Android SDK specification.

The client is configured to communicate with a Diffusion™ Server using DPT, or over a secure DPTS connection.

iOS Clients

This is a mobile client which adheres to the Diffusion™ Client API and has been written as a Objective C library.

The client is configured to communicate with a Diffusion™ Server using DPT, or over a secure DPTS connection.

J2ME Clients

This is a mobile client which adheres to the Diffusion™ Client API and has been written in Java using the J2ME specification.

The client is configured to communicate with a Diffusion™ Server using DPT, or over a secure DPTS connection.

Client Close Reason

There are many reasons why the client may be closed. These are expressed by the ClientCloseReason enumeration and described here.

CONNECTION_LOST

The connection to the client was lost - possibly dropped by the client. Recoverable. A client may be closed for may reasons that are presented as CONNECTION_LOST. During connection the connection can be lost. The server might have received a connection or reconnection request from a client already connected. The server might have received a reconnection request without a client ID. The connection may not have been authorised because the credentials are wrong. The maximum number of clients might already be connected. Once connected the connection can be lost for different reasons. If the client closes its connection while the server is writing a message to the client. With the chunked encoding based connection the HTTP response is completed by the server. If the client does not open a new request within a timeout the client will be closed. If a poll request times out and the server finds that the connection has already been closed by the client.

IO_EXCEPTION

An unexpected IO Exception occurred. Recoverable. While trying to perform an I/O operation an exception was generated. This often means that Diffusion attempted to read from a closed TCP connection. This can also happen to HTTP Comet streaming clients if there is a malformed chunk length header before a chunk of messages. This may happen if a message that is larger than the input buffer of the connector is received. The message cannot be read by Diffusion so the client is closed. When Diffusion is handling SSL connections if there is a problem encrypting or decrypting a message the client will be closed for this reason.

Copyright 2013 Push Technology

MESSAGE_QUEUE_LIMIT_REACHED

The maximum outbound queue size was reached for the client. Not recoverable. Messages sent to the client are placed in a queue. This queue has a maximum allowed size. If the queue limit is reached the client is closed and the queue discarded. The queue is intended to protect against slow patches, reaching the queue limit is taken to mean that the client cannot keep up with the number of messages sent to it.

CLOSED_BY_CLIENT

The client requested close. Not recoverable (unless TEST_RECONNECT is true).

MESSAGE_TOO_LARGE

The client sent a message that exceeded the maximum message size. The server has a maximum message size. If a client sends a message larger than this the server is unable to process it. When this happens the message is discarded and the client is closed.

INTERNAL_ERROR

An internal error occurred.

INVALID_INBOUND_MESSAGE

An inbound message with an invalid format was received. A message received by the server is not a valid Diffusion message. The server is unable to process this and closes the client that sent it.

ABORTED

The client connection was aborted by the server. This is may be because the connection was disallowed. Abort messages are also sent to clients that have unrecognised client IDs. This may be because the server closed the client previously but the client is unaware of this and tried to continue interacting with the server.

LOST_MESSAGES

Loss of messages from the client has been detected. For example, whilst waiting for the arrival of missing messages in a sequence of messages a timeout has occurred.

HTTP based transports use multiple TCP connections. This can cause the messages to be received out of order. To reorder the messages those sent to the server may contain a sequence number indicating the correct order.

If a message is received out of order there is a short time for the earlier messages to be received. If the messages are not received in this time the client is closed.

Missing, invalid or duplicate sequence numbers will also close the client for this reason.

This cannot be recovered from as the client and the server are in inconsistent states.

SERVER_CLOSING

The client was closed as part of the Diffusion shutdown process.

CLOSED_BY_PUBLISHER

A publisher initiated the client close.

CLIENT_UNRESPONSIVE

The client has either failed to respond to a ping message in a timely manner or the client has failed to open an HTTP poll for messages. The client does not appear to be receiving messages.

Copyright 2013 Push Technology

Cross-domain

What is Cross-domain

Cross-domain grants permission to comunicate with servers other than the one the client is hosted on.

Cross Domain XML File

The crosss-domain policy is defined in an XML file

A cross-domain policy file is an XML document that grants a web client—such as Adobe Flash player, Adobe Reader, Silverlight Player, etc. permission to handle data across multiple domains. When a client hosts content from a particular source domain and that content makes requests directed towards a domain other than its own, the remote domain would need to host a cross-domain policy file that grants access to the source domain, allowing the client to continue with the transaction. Policy files grant read access to data, permit a client to include custom headers in cross-domain requests, and are also used with sockets to grant permissions for socket-based connections.

For example, lets say that the Diffuison client is loaded from static.acme.com and the connection URL to the Diffusion™ client is http://streaming.acme.com, then a crossdomain.xml file will need to be loaded from static.acme.com

Crossdomain.xml is required if one of the following is true

- You are using Diffusion™ as a streaming data server and a separate Web Server which are on different domains.
- The Diffusion™ connection type is *HTTP(s)* or *HTTPC(s)* .
- You are not using a load balancer to HTTP rewrite Diffusion™ traffic

Installing the crossdomain.xml file for Flash / Silverlight HTTP request

- If you use Diffusion™ as a web server: Copy the crossdomain.xml file from the Diffusion™ install /etc folder to the root of the html folder
- If you don't use Diffusion™ as a web server: Copy the crossdomain.xml file from the Diffusion™ install /etc folder to the virtual root of the web server hosting the Diffusion™ html lib folder

By default Diffusion doesn't have crossdomain.xml installed. We shipped an example which allow all domains and all ports to access the Diffusion™ Server. This needs to be edited to include the correct security details for your installation.

Flash Security Model

How Flash interacts with remote services to establish security

If a socket based connection is to be used, i.e.Diffusion™ DPT type connection, the the flash player will try and get a policy file from the same host as you are trying to connect to but on port 843. If this port is not open through your firewalls or is not configured within the Diffusion™ Connectors the the flash player will wait 2 seconds before requesting a policy file from the same port that you are trying to connect to. If the policy file request is not responded to correctly or the policy file has restricted the connection then the flash player generates a security exception and the connection attempt will cease.

If an HTTP connection is to be used, i.e. Diffusion™ HTTP type connection, then a socket based policy file is not required but a crossdomain.xml file may be required before the Diffusion™ connection is made.

Official Adobe documentation: *Network security access restrictions (Flash)*

FlashPolicy.xml file

When is the FlashPolicy.xml used?

When a Diffusion™ *DPT* connection is used a socket connection is made, in order that the socket connection can be established a socket policy file needs to be gained from port 843 or from the port that the Diffusion™ client is trying to connect to.

Again this is part of the crossdomain schema, but this time its the *to-ports* attribute on the *allow-access-from* element that is particularly important.

Copyright 2013 Push Technology

FlashMasterPolicy.xml File

Use of the FlashMasterPolicy file

FlashMasterPolicy is used for requests on port 843. It is a normal crossdomain.xml with an extra element of

```
<site-control permitted-cross-domain-policies="master-only" />
```

The *site-control* element here specifies that only this master policy file should be considered valid on this domain

Silverlight Security Model

How Silverlight interacts with remote services to establish security

If a socket based connection is to be used, i.e.Diffusion™ DPT type connection, the the Silverlight player will try and get a policy file from the 943. If this port is not open through your firewalls or is not configured within the Diffusion™ Connectors the Silverlight player will generate a security exception and the connection attempt will cease.

If an HTTP connection is to be used, i.e. Diffusion™ HTTP type connection, then a socket based policy file is not required but a crossdomain.xml file may be required before the Diffusion™ connection is made.

Official Microsoft documentation:*Network security access restrictions (Silverlight)*

Silverlight clientaccesspolicy.xml file

When is the Clientaccesspolicy.xml used?

When a Diffusion™ *DPT* connection is used a socket connection is made, in order that the socket connection can be established a socket policy file needs to be gained from port 943.

Javascript Security Model

How Javascript interacts with remote services to establish security

The Diffusion™ Javascript Client Library, unless otherwise configured, will cascade downward through a set of 'transports' starting with WebSockets and working it's way down toward XmlHttpRequest (AKA 'XHR' or 'Ajax Long Poll'), and then finally to hidden IFrames.

Websockets, Flash and Silverlight have few security constraints, however XHR is subject to the same origin policy. Simply put, if Javascript code executes within a web page sourced from www.acme.com then it is only permitted to make XHR requests back to www.acme.com. If your diffusion server is at push.acme.com this presents a problem when only XHR is available.

The Catch-All solution

The set of web browsers in current use is both broad and heterogenous. Rather than catering to each special case browser, this approach contains all complexity to one place: the load balancer. This presumes there is a load-balancer however, though in all reasonable production circumstances this will be true. All XHR requests to Diffusion™ use a URL that starts /diffusion. Routing all such requests to one of the servers in the Diffusion™ pool will make available both regular and Diffusion™ functionality from one apparent host. This will sate even the most eccentric of web browsers.

Copyright 2013 Push Technology

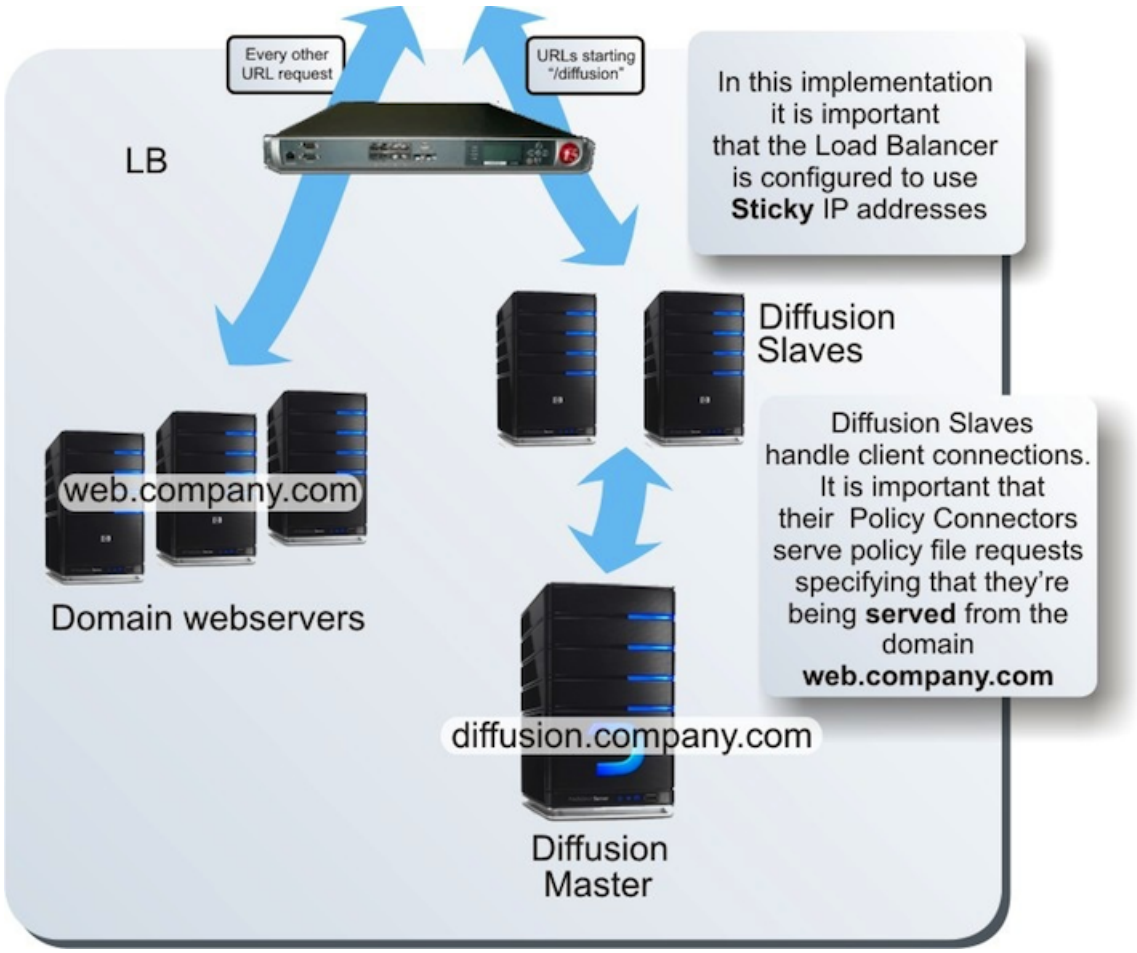

Every other URL request
URLs starting "/diffusion"
LB
In this implementation it is important that the Load Balancer is configured to use **Sticky** IP addresses
Diffusion Slaves
Diffusion Slaves handle client connections. It is important that their Policy Connectors serve policy file requests specifying that they're being **served** from the domain **web.company.com**
web.company.com
Domain webservers
diffusion.company.com
Diffusion Master

Figure 21: Using a load balancer to composite two URL spaces into one.

In circumstances where clients of Diffusion™ solutions cannot be depended upon to have a single IP address (for example: users with multiple aDSL connections, or smart-phones migrating between providers), each HTTP request made from a Diffusion™ client to a Diffusion™ server will hold a cookie named 'session' holding the unique client-id of that client. This gives load balancers an alternative means of distributing a request to one of their Diffusion™ server team. Further details can be found in the Configuration section of the manual.

Software alternatives

For test and development purposes a hardware load balancer may be an expensive means of compositing the URL-spaces of two (or more) web servers into one. Alternatives such a mod_proxy for the Apache web server, and an ISAPI_Rewrite tool can be employed to achieve the same for a lower (or zero) price-tag. Diffusion™ can be configured to run as a servlet with Tomcat. Details on this are avail in the "Web Server Installation" section of the Diffusion™ manual

CORS

Cross-Origin Resource Sharing is a standard formed to address circumstances where www.acme.com uses XHR to access resources on alternate host push.whatnot.com, and aims to provide sensible constraints and avoid a free-for-all.

Client-Side

Include the "XHRURL" attribute in the arguments to the DiffusionClientConnectionDetails constructor. For example..

```
var connectionDetails = {
          debug : true,
          onDataFunction : onDataEvent,
          XHRURL: "http://www.whatnot.com:8080"
```

Copyright 2013 Push Technology

```
            topic : 'SYMBOLS/QUOTES/NIFTY~INDEX',
}
```

Server Side

CORS filtering is governed on the server side using the cors-origin attribute found in *etc/WebServer.xml* . By default this is a very permissive `.*` regular expression, and must be set to something more specific in production. In the above example, `push.whatnot.com` will limit requests to `push.whatnot.com` to only those from `www.acme.com` Full details on this feature are found in the Web Server section of the Diffusion™ manual.

Discussion

CORS support is currently lacking from Internet Explorer however, making it useless in most internet-facing production scenarios.

Browsers supported

Some of the client libraries are intented to be run within browser environments. These environments are not uniform and broswers may have functional limitations. It is important to be aware of these limitiation when developing for compliance with target browers.

Diffusion™ supports the latest release of the above browsers based on the original Diffusion™ release date. It can use most commercial browsers and their variants, although all Diffusion™ API protocols may not be available.

Table 16: Browsers Supported

Browser	Version
Google Chrome	25.0
Mozilla Firefox	19.0
Microsoft Internet Explorer	9.0
Apple Safari	6.0

- Push Technology runs full regression tests on the browsers and versions documented above for every patch release.
- Push Technology also carries out basic validation testing on the latest versions of these browsers but full support is only available at the next minor release and thus support for untested browsers is provided on a best effort basis.
- Support for older versions of browsers is also provided on a best-effort basis, unless specified otherwise.
- Support for other browsers is also provided on a best-effort basis.

Browser's plugins (Flash and Silverlight)

Diffusion™ only supports official Flash and Silverlight players.

Table 17: Browser Plugins

Plugin's player	Version
Adobe Flash Player	11.6
Microsoft Silverlight	5.1

For details on the Operating Systems and Browsers supported by the Silverlight plugin, refer to the Microsoft-maintained Silverlight 'System Requirements' section *here.*

Copyright 2013 Push Technology

Browser Limitations

Some browsers cannot use all the Diffusion™ protocols or features.

Websocket

WebSocket is a *IETF* protocol used by Diffusion™ to provide a full-duplex communication channel over a single TCP connection.

Table 18: Websocket

IE	Firefox	Chrome	Safari	Opera	iOS	Android
6.0 (Not supported)	15.0	22.0	5.1	12.0	3.2 (Not supported)	2.1 (Not supported)
7.0 (Not supported)	16.0	23.0	6.0	12.1	4.0 (Not supported)	2.2 (Not supported)
8.0 (Not supported)	17.0	24.0		12.5	5.0 (Not supported)	2.3 (Not supported)
9.0 (Not supported)	18.0	25.0			6.0	3.0 (Not supported)
10.0	19.0					4.0 (Not supported)
						4.1 (Not supported)
						4.2 (Not supported)

Cross-origin resource sharing (CORS)

Allows resources to be accessed by a web page from a different domain.

Table 19: Cross-origin Resource Sharing (CORS)

IE	Firefox	Chrome	Safari	Opera	iOS	Android
6.0 (Not supported)	15.0	22.0	5.1	12.0	3.2	2.1
7.0 (Not supported)	16.0	23.0	6.0	12.1	4.0	2.2
8.0 (Not supported)	17.0	24.0		12.5	5.0	2.3
9.0 (Not supported)	18.0	25.0			6.0	3.0
10.0	19.0					4.0
						4.1
						4.2

Cross-origin resource sharing uses HTTP headers to allow the Diffusion server to indicate if it accepts traffic from web pages served from other servers. When a CORS request is made Diffusion must respond with certain response HTTP headers so the browser will treat the request as successful. CORS requests may result in the browser sending a preflight request to Diffusion using the OPTIONS method to determine if the origin, headers, and methods of the

Copyright 2013 Push Technology

request it is about to make as permitted. Diffusion will automatically respond with the correct values for headers and methods but the actual request will not be made until the preflight request succeeds. The allowed origins can be configured in the *client-service* of the WebServer.xml.

Number of connections per domain

All the browsers have a limited number of HTTP connections within the same domain name. This restriction was defined in the HTTP Specifications (RFC2616). Initially, browsers could not maintain more than 2 connections with any server or proxy, but nowadays this number is bigger. Most of the modern browsers allow 6 connections per domain.

Table 20: Max Connections per Domain

Browser & Version	Max Connections
IE 6.0	2
IE 7.0	2
IE 8.0	6
IE 9.0	6
IE 10.0	8
Firefox 15.0	6
Firefox 16.0	6
Firefox 17.0	6
Firefox 18.0	6
Chrome 22.0	6
Chrome 23.0	6
Chrome 24.0	6
Chrome 25.0	6
Safari 5.1	6
Safari 6.0	6
Opera 12.0	6
Opera 12.1	6
Opera 12.5	6
iOS 3.2	6
iOS 4.0	4
iOS 5.0	6
iOS 6.0	6
Android 2.1	4
Android 2.2	4
Android 2.3	9
Android 3.0	6
Android 4.0	6
Android 4.1	6

Copyright 2013 Push Technology

Diffusion™ protocols like HTTP, HTTPS, HTTPC and HTTPCS use up to two simultaneous connections per Diffusion™ client. It is important to understand that the maximum number of connections is per browser and not per browser tab. Attempting to run multiple clients within the same browser may cause this limit to be reached.

The HTTP 1.1 Protocol states that *single-user clients should not maintain more than two connections with any server or proxy*. This is the reason behind these browsers limits. Modern browser are less restrictive than this, allowing a larger number of connections. The RFC does not specify what should be done to prevent the limit being exceeded. Connections may either be blocked from opening or existing connections closed.

Reconnection can be used to maintain a larger number of clients than would usually be allowed. When TCP connections for HTTP requests are closed the Diffusion will send another HTTP request which the server will accept. You must bare in mind cases where Diffusion tries to write a response to closed polling connections before the client can restablish them, this results in an IO Exception and Diffusion server will close the client unless reconnection is enabled. This would result in aborting the client when it tries to reestablish the poll.

Another way to get around browser limits is by providing multiple subdomains. Each subdomain is allowed the maxium number of connections. Using numbered subdomains a client can pick a random subdomain to connect to. Where the DNS server allows subdomains matching a pattern to be resolved as the same server, tab limits can mitigated substantialy.

Plugin Limitations

When plugins such as Flash or Silverlight need to make HTTP requests they go through the browser instead of making them directly. This means they do not avoid browser limits on the number of connections and must wait for the browser to make the request. There is an additional layer between the client and the connection.

Browser Buffering

When streaming, such as with HTTPC connections, the web browser may not release the data it recieves imediately. Browsers may require that a minium amout of data to be received before passing it onto the client. After this minimum amount has been received future messages will be passed by the browser as soon as it receives them. This may cause problems when there is a delay between the inital topic load and delta messages. To handle this the first message sent over a HTTPC poll will be padded with null bytes to fill this buffer. The amount of padding sent can be configured with the comet-initial-message-padding configuration item (see *WebServer - Client Service*). The amount of padding required will vary between browsers.

Protocols Supported

Each client supports varying transports. A table of the supported transports for each client is presented here.

The APIs table describes a more detailed list of the implemented protocols. Some protocols have been partially implemented and are not supported. Support for J2ME and C is on a best effort basis.

The JavaScript client is only fully supported on certain browsers. Best effort support is provided for other browsers but the software/hardware combination may not be reproducible, particularly for mobile browsers. Refer to the *Browsers Supported* for detail on the supported browsers.

You should refer to the *API documentation* for more information on what is possible with the different client APIs.

You should refer to the *Protocol Section* for more information on the protocols. Here is a list of the protocols supported for each client:

Client	DPT	DTPS	WS	WSS	HTTP Full Duplex	HTTPS Full Duplex	HTTP Chunked Streaming	HTTPS Chunked Streaming	HTTP Polling	HTTPS Polling
Java	Supported	Supported	Supported	Supported	Supported	Supported				
.NET	Supported	Supported	Supported	Supported	Supported					

Copyright 2013 Push Technology

Client	**DPT**	**DTPS**	**WS**	**WSS**	**HTTP Full Duplex**	**HTTPS Full Duplex**	**HTTP Chunked Streaming**	**HTTPS Chunked Streaming**	**HTTP Polling**	**HTTPS Polling**
JavaScript	Supported (By Flash/ Silverlight)	Supported (By Flash/ Silverlight)	Supported	Supported			Supported (By Flash)	Supported (By Flash)	Supported	Supported
Flash	Supported	Supported					Supported	Supported	Supported	Supported
Silverlight	Supported	Supported							Supported	Supported
iOS	Supported	Supported								
J2ME	Supported	Supported								
Android	Supported	Supported								
C	Supported									

Copyright 2013 Push Technology

Chapter 10

Topics

Topics:

- *Topic Basics*
- *Topic Hierarchy*
- *Topic Naming*
- *Topic Aliasing*
- *Creating Topics*
- *Topic Subscription*
- *Topic Loading*
- *Topic Data*
- *Topic Selection*
- *Topic Fetch*
- *Topic Sets*
- *Topic Attachments*

Diffusion™ publishes messages to topics, to which clients may subscribe. All Clients subscribed to a Topic receive all Messages published to the Topic. The Topic is therefore a fundamental part of Diffusion™.

Topic Basics

This section introduces the concept of Topics within Diffusion™.

What is a Topic?

Diffusion™ is essentially a 'publish/subscribe' Message Broker.

Publish/subscribe (or pub/sub) is an asynchronous messaging paradigm where senders (publishers) of messages are not programmed to send their messages to specific receivers (subscribers). Rather, published messages are characterized into topics, without knowledge of what (if any) subscribers there may be. Subscribers express interest in one or more topics, and only receive messages that are of interest without knowledge of what (if any) publishers there are. This decoupling of publishers from subscribers allows for much greater scalability and a more dynamic network topology.

In Diffusion™ a Topic has state, unlike the queues/topics of a message-broker. It is a named object that is registered (or added) in a Diffusion™ Server by a Publisher and enables not only basic pub/sub messaging but also a number of other features which are introduced below.

How are Topics Used?

Topics can be used in a number of ways.

The main ways in which Topics are used within Diffusion™ are as follows:

Publishing to all subscribed Clients

The Publisher publishes Messages to the Topic. Clients subscribe to Topics and each Client that is subscribed to a Topic will receive all Messages published to that Topic.

Publishing to all but one subscribed Client

A Publisher can 'publish' a Message to a Topic but excluding a specified Client.

Sending Messages to individual Clients

A Publisher can 'send' a Message to an individual Client (or group of Clients) which the Client receives like any other Message delivered for the Topic.A Client can send a Message to a Topic which in turn is routed to the Publisher.

Client to Publisher Communication

A Client can send a Message to a Topic which in turn is routed to the Publisher.

Client Fetch

A Client can retrieve the current state of an existing Topic without subscribing to it using a 'fetch' request. When a Client attempts to fetch the state of a Topic that does not exist then a clientFetchInvalid notification occurs which allows the Publisher to send a response to a fetch request (using Client.sendFetchReply) for a Topic that does not even exist if that is what is required.

Receiving Event Data

An 'Event Publisher' can connect to a Diffusion™ Server and send Messages to a Topic. Such Messages will be routed to the Publisher providing the Topic.

Distributing Publishers

Publishers can be distributed by means of a Publisher connecting to another Diffusion™ Server and subscribing to Topics as if it were a Client. The communication available between Publishers is then provided via Topics in exactly the same way as communication between Publishers and Clients in general.

Copyright 2013 Push Technology

Topic Hierarchies

Topics are arranged in hierarchies

In the simplest case Topics are flat in structure. For example you could have a single Publisher providing three Topics called "A", "B" and "C" which would be subscribed to individually.

Topics may also be organised hierarchically so that Topics have subordinate Topics and form a Tree of Topics. This allows for much more control over the Topics, for example it is possible to subscribe to branches of the tree rather than individual Topics, e.g. "foo/bar//". To better manage this control, subordinate Topics cannot be created by Publishers that did not create the parent Topic. This prevents subscriptions to branches of the tree from being handled by multiple different Publishers.

Subscription

Subscription refers to a Client's association with a Topic

'Subscription' is the term used to describe a Client registering an interest in a Topic such that it receives all Messages published to that Topic.

A Client may subscribe to any number of Topics.

When a Client subscribes to a Topic typically the first Message it will receive on that Topic will be a 'Topic Load' (see below) which provides the current data state of that Topic. It will then receive all updates to the Topic as 'Delta' Messages.

A Client may subscribe to a Topic using an explicit Topic name or may subscribe to many Topics using a Topic Selector.

A Client may subscribe to a Topic that does not exist. It will not be informed in any way. However, Publishers can be notified that a Client has attempted to do this. When a Publisher creates a new Topic then it has the option to subscribe any Clients to it that may have tried to subscribe to it previously. When Clients subscribe to Topics before they are registered and are then susbcribed later when the Topic is registered, it is called 'pre-emptive', or 'future' subscription.

When a client requests subscription to a topic, that subscription is validated in the following order;

- Authorisation Handler
- Subscription Policy (Blacklist/Whitelist)
- Topic Subscription Handler

Clients may resubscribe to Topics that they are already subscribed to in order to force re-transmission of a Topic Load, but without being doubly subscribed.

Clients may unsubscribe from Topics at any time.

Topic Loading

The initial state of a Topic is referred to as the 'Topic Load'

When a Client subscribes to a Topic the Publisher is notified. At this point it is usual for the Publisher to provide the current data state of the Topic in the form of a Topic Load Message. This act of providing the initial Topic Load data is know as 'Topic Loading'.

Topic Data

Topic Data is the Data state associated with a Topic

In the broadest sense, Topic data is best thought of as the data associated with a Topic and maintained by the Topic provider. This data has an initial state and may then be updated causing Delta Messages to be sent to Clients to notifiy the changes to the data. New Clients subscribing to a Topic would typically be sent the current state of the Topic's data as a Topic Load (see above).

Copyright 2013 Push Technology

To simplify the management of Topic Data the Java API provides built in mechanisms for maintaining and updating Topic Data and reading and generating various types of Message content. For more information see the Topic Data section.

Topic Attachments

A Topic may have any type of Object attached to it for prcessing convenience

The Java API allows an object to be attached to a Topic. This might be used to associate some data handling object with the Topic when the Topic Data feature is not appropriate.

Topic References

A alternative name or peice of text may be associated with each Topic

Often Topic names may not be very meaningful. The Java API allows a 'reference' to be set for a Topic which might be used in situations where the Topic name might be displayed.

Topic Listeners

Object may be nominated to receive Topic events and process inputs for Topics

As Topic Messages arrive at an application (e.g. a Publisher) they are routed to methods to handle them. However, in addition to the normal recipient method it is possible to specify additional 'Topic Listeners' which will handle Messages for specified Topics. Such a listener may handle the messages for only one Topic or for many different Topics (specified using topic names regular expression patterns).

For example, a Publisher can add different listeners to handle Message from Clients, Event Publishers or Servers or a Client can add listeners to handle different Messages from servers.

Topic Listeners allow for the processing for particular Topics to be easily encapsulated.

Topic Listeners are invoked in the order that they are declared and before any fixed listener interface. Any Topic Listener may choose to 'consume' the message which means that it would then not be passed to any other listeners.

Topic Providers

A Topic Provider is used to create and manage Topics.

A Topic Provider is able to create and destory Topics. It is not nessasary to be a Topic Provider to send messages to a Topic but a Topic Provider can be used to create a `TopicMessage` object. A Topic Provider owns the Topics that it creates. A Topic Provider cannot create a subordinate Topic to a Topic that it does not own. An `TopicInvalidException` is thrown if you attempt to add a topic that is owned by another Topic Provider. Topic Providers can remove Topics that they do not own. Every Topic knows it's Topic Provider and can get it with `getTopicProvider` .

Publishers are the used to implement Topic Providers. Ownership of Topics is intended to enforce a single publisher handling subscriptions and initial data loads for an enitre branch of the Topic Tree. The owner of a Topic is distinct from the parent of a Topic. The owner will always be a Publisher.

Topic Hierarchy

In a very simple case there might be only one Publisher deployed in a Diffusion™ Server which has a small number of discrete Topics. At the other extreme there may be many Publishers, each with many (perhaps thousands of) Topics. The number of Topics is largely to do with how it is decided to model the data that will be published for an application.

When there are very few Topics it may be easy enough to organise them in a flat Structure as follows:-

Copyright 2013 Push Technology

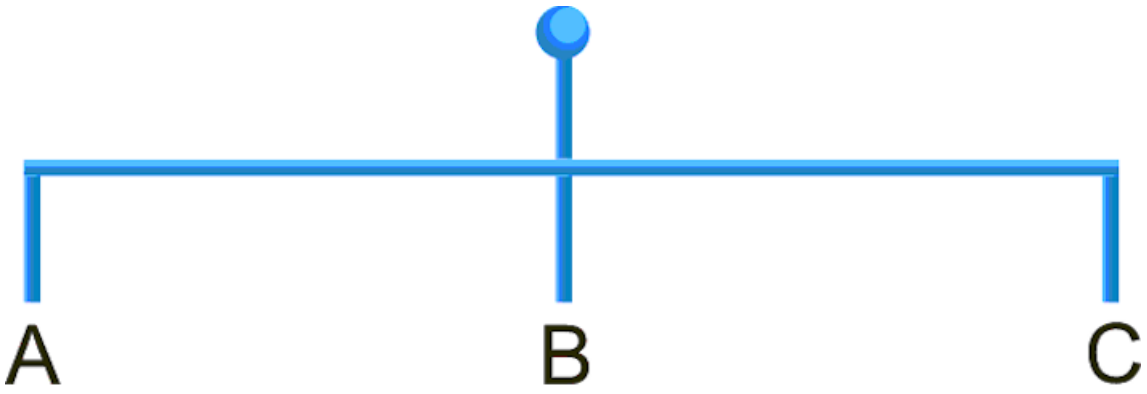


Figure 22: Flat Structure

However, when there are a large number of Topics it becomes more difficult to handle them as the Publisher has to register and load each one and the Client has to subscribe to each one. Also it may be difficult to map complex data structures onto Topics. Diffusion™ allows Topics to be organised hierarchically to simplify the handling of those Topics and also the mapping of data to Topics.

An example of a hierarchical Topic structure could be as follows:-

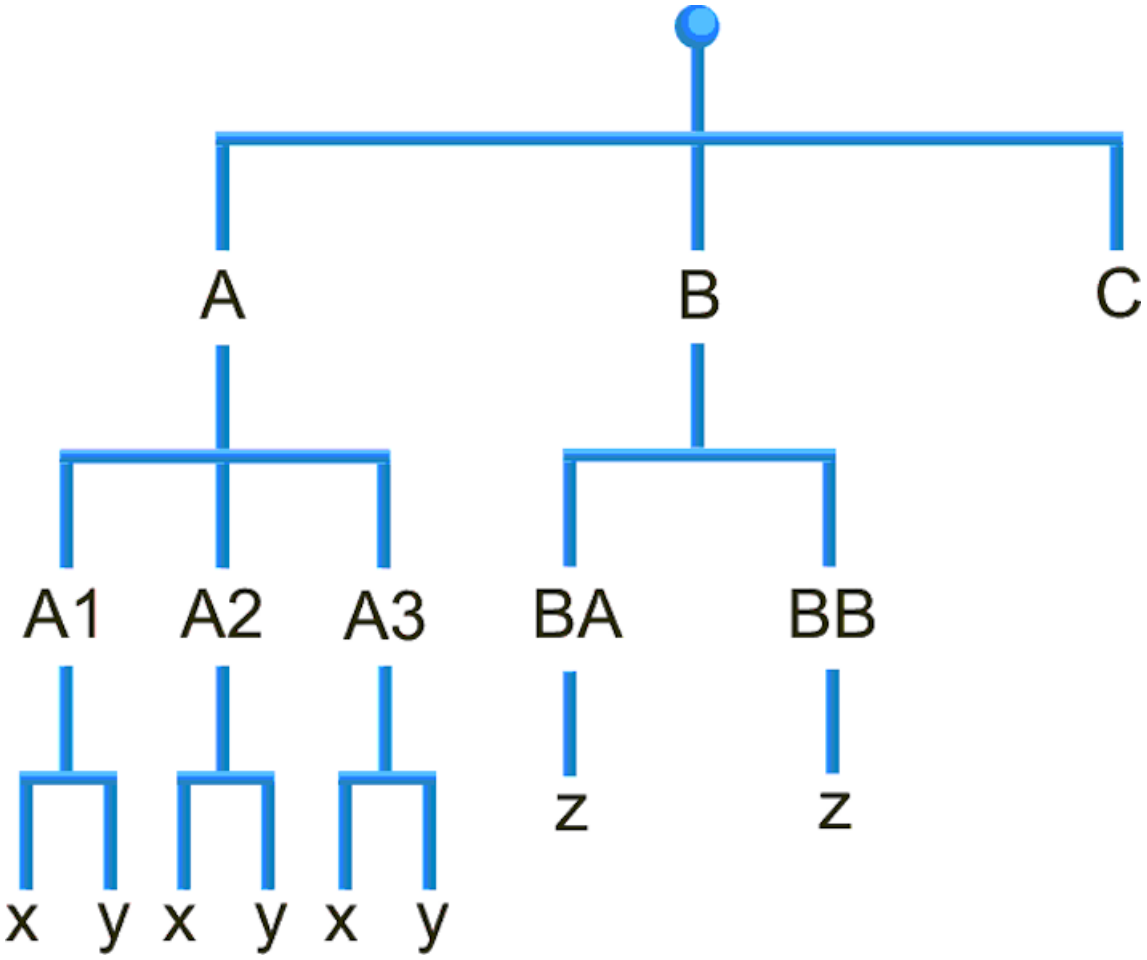


Figure 23: Hierarchical Topic Structure

In the above case the Topics A, B and C could be provided by different Publishers or all by the same Publisher. The difference from the first examples is that A and B have subordinate Topics allowing data to be organised in a more

Copyright 2013 Push Technology

structured way. The Topics A, B and C will be owned by the Publisher providing them and the subordinate Topics will only be creatable by the owning Publisher. A Publisher cannot create the Topic A1 if does not own A.

The Topic Tree

The hierarchy of Topics that is present on any one Diffusion™ Server is known as the 'Topic Tree'. Inside the Topic Tree is a single 'root' node with subordinate 'Topic' Nodes (so every node in the tree is a Topic other than the root).

The root node of the Topic Tree may be obtained using the

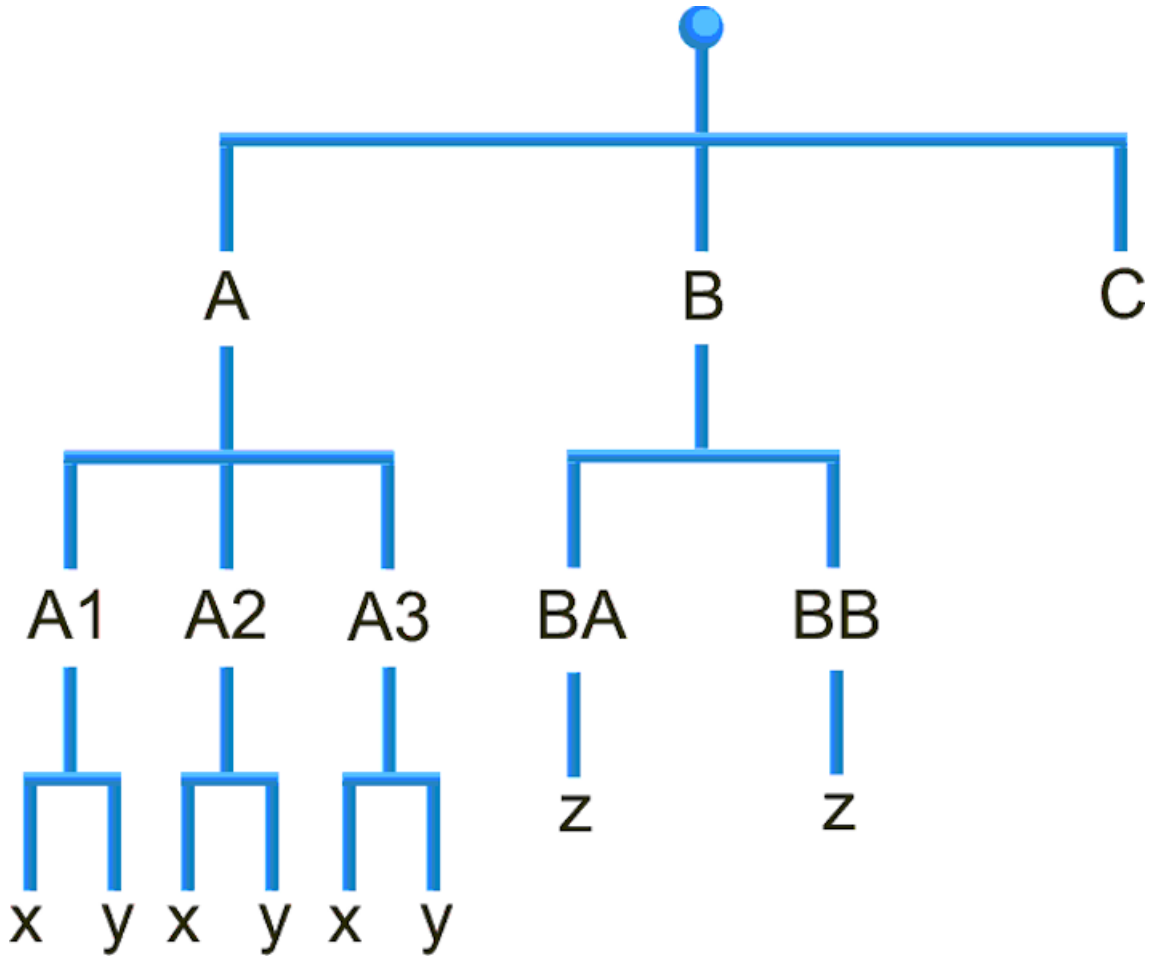
```
Publishers.getTopicTree
```

method which returns a TopicTree. Every subordinate node is a Topic. All nodes (root and Topics) are of type TopicTreeNode. These classes provide comprehensive mechanisms for navigating the tree structure. The TopicTree node has a fast Topic lookup facility in its getTopic method.

Each TopicTreeNode is owned by the Topic Provider that created it. The ownership of a Topic is distinct from the parent of a Topic. They are related by controlling which Publisher can create a child Topic Node on the Topic Tree.

Referencing Hierarchic Topics

A hierarchic Topic may be accessed using its 'path' or full hierarchic name which is represented as the name of each Topic node within it, hierarchy separated by '/'. It is this name that is used at subscription and it is this name that is carried within Messages.



Copyright 2013 Push Technology

For example, in the above tree the names of the Topics in the branch headed by the Topic called 'A' are:-

`AA/A1A/A1/XA/A1/YA/A2A/A2/XA/A2/YA/A3A/A3/XA/A3/Y`

Referencing Segments of the Topic Tree

Topics may be referenced individually by their full path name but in many situations there is also the facility to refer to branches of the Topic tree or 'fuzzy match' with Topic names. This is fully described in 'Topic Selection'.

Subscribe all subordinates

For example, to subscribe to Topic 'A' above and all its subordinates using the Java Client API

```
ExternalClientConnection connection =
 new ExternalClientConnection(this,"dpt://MyHost:8080");
connection.connect();
connection.subscribe("A//");
```

Subscribe only children

Or to subscribe to only the Topics (X and Y) subordinate to the A2 Topic

```
connection.subscribe("A2/");
```

Topic Naming

This section covers all aspects of Topic naming including recommendations of how Topics should be named.

A Topic name may be made up of any number of Unicode characters but must not contain any of the restricted characters mentioned below. When hierarchic Topics are in use the Topic Name is made up of the names of all Topics in its hierarchy separated by "/".

Topic Naming - Restricted Characters

The following characters are not permitted in Topic (node) names:-

Table 21: Restricted Characters

Character	Reason for restriction
/	This is a node name separator and therefore cannot occur in a Topic node name.
[]\^$.|?*+()	These are all metacharacters used in regular expressions and therefore any Topic String that contains any of these characters will be assumed to be a Topic Selector. These characters can therefore not be used in Topic node names.
Control/Reserved	No characters with a hexadecimal value of less than 0x2D. This includes some punctuation characters such as ','

Copyright 2013 Push Technology

Character	Reason for restriction
Whitespace	No characters defined as whitespace in Java (as indicated by the isWhiteSpace method of the Java Character class).

Topic Naming Recommendations

Although all unicode characters (other than the restricted ones mentioned above) are supported to allow for language variations it is highly recommended that only alphanumeric characters are ever used in Topic Names with. '-' (hyphen) or '_' (underline) used as break characters.

Generally speaking, Topic names should be kept short as every Topic Message has to carry the full Topic Name which could potentially have significant Message size implications. However, the optional Topic aliasing feature can significantly reduce the size of the Topic names transmitted in Messages which can be very important, especially when hierarchic Topics are in use.

Topic References

Within a Publisher a Topic may be assigned a more meaningful reference than its Topic name. This can be helpful in identifying the Topic in logs etc, especially when short unique Topic names are used.

A Topic reference may be set using the setReference method and obtained using getReference.

Topic Aliasing

How to save space in Messages

It is important to remember that the Topic name forms part of each Message header, so the smaller the Topic name the better. However, when hierarchic Topics are in use it may be difficult to keep Topic names short.

Topic aliases are a mechanism for automatically using short Topic names in all Delta Messages. It is an optional feature, but turned on by default.

The use of Topic aliases can save a significant amount of network traffic, however there is a trade off of a small processing cost which can reduce the Message throughput.

Topic aliasing may be turned on or off for a Publisher using a *etc/Publishers.xml* entry. For an Event Publisher it can be turned on or off using a `setTopicAliasing` method.

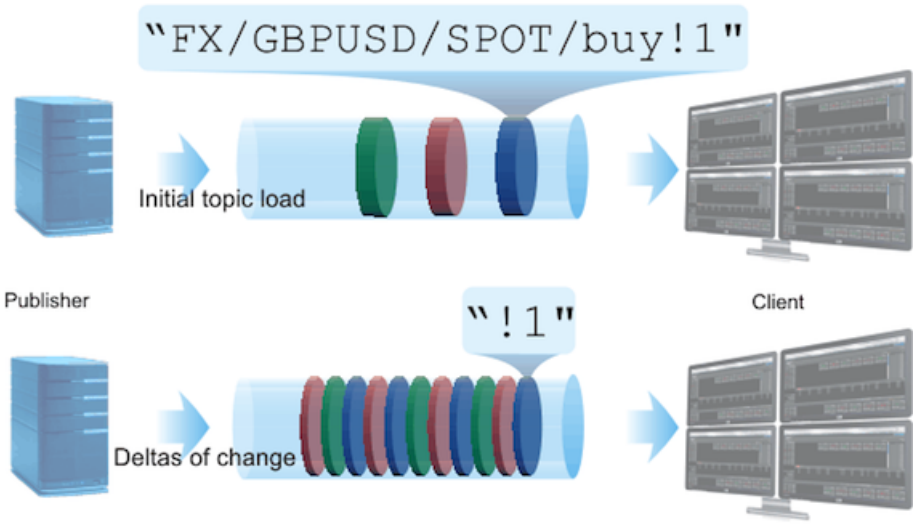


Figure 24: Topic Aliasing

Copyright 2013 Push Technology

When turned on, the creation of Topic Load Messages causes short Topic aliases to be generated for all Topic names greater than 4 characters in length. Delta messages will then only contain the Topic alias thus saving space in each Message sent.

Warning - if aliasing is turned on, then creating the Load message without sending it to Clients will still result in Delta messages being aliased - this can lead to failures at the client.

It is possible to override the publisher aliasing requirement on a per Topic basis by creating the Topic with a TopicDefinition and explictly specifying the aliasing property.

The use of Topic Aliases is totally transparent to the API where Topic names are always presented in full. However, the diagnostic display of Messages may show an alias (a short id prefix by a `!` character).

Note that if Topic Load Messages are not used for a Topic then an alias will not be generated for that Topic therefore the saving will only be achieved for Topics that send a Topic Load Message to the Client on subscription. Likewise, unless the first Message sent by an Event Publisher for any Topic is a Topic Load then subsequent deltas will not use aliases.

Creating Topics

How are Topics Created

Topics are created by Publishers to make them available for Clients for subscription.

Topics may be created on behalf of a Publisher in the xml configuration files or by creating them dynamically from within the Publisher.

Configuring Topics

How to Specify Topics in Publishers.xml

When a Publisher provides a relatively small number of Topics (or you want it to start with a small number of Topics) then the one way of creating topics is by configuring them for the Publisher in *etc/Publishers.xml*

One or more Topics can be named in the configuration and these Topics will then be automatically added during Publisher startup.

Any Topics specified in this way would also be re-added if the Publisher were stopped and restarted.

Topics added in this way are available for subscription to immediately and therefore the data required for the Topic would need to be set up during subscription.

This method of specifying Topics via configuration is not compatible with the use of Topic Data and therefore has limited use.

Dynamically Adding Topics

How to Create Topics from within a Publisher

The most flexible way of creating Topics is dynamically within the Publisher. This way a Publisher need not start up with a full set of Topics but can add them as and when required.

This method can still be used along with configured initial Topics. For example, you may want the Publisher to start with some fixed 'control' Topic and then others are added according to what is required.

The following set of methods are available to a Publisher for adding Topics:-

Table 22: Methods for Adding Topics to Publishers

1	`addTopic(String)` `addTopic(String,TopicTreeNode)`	This is the simplest method where a top level Topic of a given name is registered.

Copyright 2013 Push Technology

		This can also be used to add a hierarchic Topic by specifying a hierarchic name and any missing intermediate Topics would also be registered. It is however more normal to add hierachic Topics using the second method where the parent Topic is specified.
2	`addTopic(String,TopicData)` `addTopic(String,TopicTreeNode,TopicData)`	As '1' but allowing TopicData to be associated with the Topic. The TopicData must have been pre-created using `TopicDataFactory` and configured as required.
3	`addTopic(String,TopicDefinition)` `addTopic(String,TopicTreeNode,TopicDefinition)`	As '1' but allowing all details of the Topic and its TopicData to be specified. This is the most powerful method of creating Topics and can be used to create Topics using the definition from another Topic. It is not possible to create Remote Control Topics or their subdomain Topics using this mechanism.
4	`addTopics(TopicSet)` `addTopics(TopicSet,TopicTreeNode)`	Allows a set of Topics to be registered. Topic Data or attachments may not be specified when multiple Topics are registered at once.

Topics themselves also have similar methods so that child Topics can be easily added without having to specify the full hierachic Topic names as in the above examples.

Removing Topics

The effects of Removing Topics

When a Publisher is stopped all of its Topics are automatically removed, however individual Topics may be removed at any time by a Publisher.

When a Topic is removed all Clients that are subscribed to it are automatically unsubscribed.

When a Topic is removed then all Topics that are subordinate to it in the Topic Hierarchy are also removed.

Topic Subscription

'Subscription' refers to a Client registering an interest in a Topic such that it will automatically receive all Messages published to that Topic.

A Client may subscribe to any number of Topics and a Topic may be subscribed to by any number of Clients.

Subscribing to a Topic also allows a Client to send Messages to the Publisher of that Topic.

Topic subscriptions totally decouple Clients and Publishers. A Client has no knowledge of the Publisher(s) of the Topics it subscribes to.

Copyright 2013 Push Technology

Subscription on Connection

A Client may subscribe to any number of Topics when it connects to the Diffusion™ Server, or it can connect with no Topics at all and subscribe later.

The way in which Topics are specified on connection varies according to the Client API in use. The following example shows subscription on connection in the Java Client API:-

```
ExternalClientConnection connection =
 new ExternalClientConnection(this,"dpt://localhost:8080");
connection.connect("TopicA","TopicB");
```

Note that the connect method is a varargs method so any number of Topics can be specified. There is also a variant that takes a TopicSet.

Ad Hoc Subscription

A Client may subscribe to any number of additional Topics after it has connected.

The following example shows subscription after connection in the Java Client API:-

```
ExternalClientConnection connection =
 new ExternalClientConnection(this,"dpt://localhost:8080");
connection.connect();
//...
//...
connection.subscribe("TopicA","TopicB");
```

Note that the subscribe method is a varargs method so any number of Topics can be specified. There is also a variant that takes a TopicSet.

Effect of Subscription

When a Client subscribes to a Topic then the Publisher of that Topic will be notified via its subscription method. Typically the Publisher will then send a 'Topic Load' message containing the current state of the Topic's data to the Client.

From that point on all (Delta) Messages published by the Publisher to the Topic are consequently sent to the Client, until the Client either disconnects or unsubscribes from the Topic.

Clients of different types receive Messages in different ways. In the Java Client API a message is received via the messageFromServer method on the ServerConnectionListener interface. Different listeners may be assigned to different Topics as required.

Subscribing to Topics that do not Exist

If a Client attempts to subscribe to a Topic that does not exist then any Publisher that is listening for Client notifications will be notified via the `clientSubscriptionInvalid` method. This allows a Publisher to provide Topics on demand as (assuming it is a Topic it should be providing) it can then add the Topic and subscribe the Client to it using the `Client.subscribe` method.

When a Client subscribes to a Topic that does not exist the Client is not notified in any way and will not be aware that this has happened.

Subscription Using Topic Selectors

Copyright 2013 Push Technology

A Client may subscribe to specific Topics but may also use Topic Selection when subscribing to subscribe to many Topics at once. Depending upon the selector format the Client can easily subscribe to a branch of the Topic tree or to a filtered view of the Topics.

Note that when selectors are used Publishers are not notified of attempts to subscribe to Topics that do not exist.

Subscribe All Subordinates

So, in the Java Client API the Client could subscribe to a Topic called "A" and all of its subordinate Topics as follows:-

```
connection.subscribe("A//");
```

Subscribe to Top Level Topic

Subscribe to all Topics named "B" that are subordinate to any top level Topic that starts with "A" (e.g "Accounts/B" or "Admin/B") as follows

```
connection.subscribe("A+/B");
```

Subscribe Multiple Selectors

Multiple selectors may be supplied and mixed with real Topic names, for example

```
connection.subscribe("A//","MyTopic","A+/B");
```

Subscription using Topic Sets

At times it may be more convenient to use Topic Sets to specify the Topics to be subscribed to.

the following example shows how this may be done in the Java Client API:-

```
TopicSet topicSet = new TopicSet("A//","MyTopic");
//.
topicSet.add("A+/B");
connection.subscribe(topicSet);
```

Forced Subscription

A Publisher may force Clients to become subscribed to a Topic even if the Client has not requested subscription to the Topic.

Forced Subscribe Individual Client

To force subscribe an individual client to a Topic a Publisher may use the `Client.subscribe` method with the force flag set to 'true'. The following example shows a Publisher that is listening for Client notifications force subscribing all Clients that connect to its Topic:-

```
public void clientConnected(Client client) throws APIException {
 Topic topic = getTopic("MyTopic");
 client.subscribe(topic,true);
```

Copyright 2013 Push Technology

```
}
```

Forced Subscribe By Topic Set

The subscription could equally have been done using the Topic name, or even multiple Topic names by means of a TopicSet as follows:-

```
client.subscribe(new TopicSet("MyTopic","AnotherTopic"),true);
```

Forced Subscribe All Clients

It is also possible to subscribe all connected Clients to a Topic from within a Publisher as follows:-

```
Topic topic = getTopic("MyTopic");
subscribeClients(topic,true);
```

Pre-emptive Subscription

'Pre-emptive subscription' describes the facility whereby a Client can subscribe to a Topic before it exists, so that if the Topic does come into existence then the Client will automatically become subscribed to it.

The way in which this works is that all subscriptions that have been requested by a Client are remembered in the order that they were issued and applied to a new topic name in that order to decide whether it should be subscribed. So, if a Client requested subscription to the (non existent) Topic "Accounts", and then used a selector to unsubscribe from all Topics starting with "A", then when Topic "Accounts" is created and subscribed the Client would NOT become subscribed to it.

Manual Pre-emptive Subscription

Publishers may use the `subscribeClients` method (with the force flag set to 'false') on either the Publisher or Topic class after creating a new Topic. This causes any Client that has tried to subscribe to the Topic explicitly, or has subscribed with a Topic selector that would include the new Topic, to become subscribed to it.

The following example shows a Publisher subscribing any Clients that may have previously registered interest in a newly created Topic:-

```
Topic topic = addTopic("NewTopic");
subscribeClients(topic,false);
```

Automatic Pre-emptive Subscription

It is possible to indicate that for any Topic tree node (i.e. Topic or root of Topic tree) that when a new child (or descendant) Topic is added then Clients that have registered a pre-emptive interest in the Topic will automatically be subscribed.

The following example shows auto subscribe being set for a new Topic. When any sub-Topic is subsequently created beneath the Topic then any Clients that have registered an interest will be automatically subscribed.

```
Topic topic = addTopic("NewTopic");
```

Copyright 2013 Push Technology

```
topic.setAutoSubscribe(true);
```

Note that the call to `setAutoSubscribe(true)` has the effect of subscribing the Topic itself (and any subordinates) at the times of calling and so a call to subscribeClients as shown in the manual example is not necessary.

Automatic Pre-emptive Subscription All Topics

To set automatic subscription for all Topics you can call for the Topic tree:

```
Publishers.getTopicTree().setAutoSubscribe(true);
```

Subscribing Clients Individually

If required then pre-emptive subscription can also be achieved on an individual Client basis using Client.subscribe() specifying force=false. There are variants of this method that take both a Topic and a TopicSet.

Re-Subscription

If a Client subscribes to a Topic that it is already subscribed to then the Topic Load is repeated. This would typically cause a new Topic Load Message to be sent to the Client, but would not cause a Client to be doubly subscribed.

Unsubscribing

A Client may unsubscribe from a Topic to which it is subscribed to at any time and from that point on will no longer receive Messages for that Topic.

The method for unsubscribing varies for different APIs. In the Java Client API the ExternalClientConnection class has unsubscribe methods that take a list of Topic names (or selectors) or a TopicSet.

Force Unsubscribing

As with subscribing, it is possible for a Publisher to force Clients to be unsubscribed from Topics. The Client has methods for unsubscribing individual Topics or a TopicSet. Using a TopicSet it is possible to specify selector patterns to unsubscribe.

The following example shows a Publisher unsubscribing a Client from a Topic on notification of its queue limit being reached:-

```
public void clientQueueThresholdReached(
 Client client,
 boolean upper,
 int threshold) throws APIException {
  if (upper) {
   client.unsubscribe(getTopic("NewsTopic"));
  }
}
```

Subscription Handlers

It is possible to assign a 'subscription handler' to a Topic. Such a handler will be be called whenever a Client attempts to subscribe to a Topic. It is called after any other authorisation processes in order that it may perform Topic specific

Copyright 2013 Push Technology

authorisation for a particular Client's request to subscribe to the Topic. It is also possible to delegate authorisation to some asynchronous process and defer subscription until a response is received.

Topic Specific Authorisation

Simple authorisation using a Subsctiption Handler

The following example shows a subscription handler being used to prevent mobile clients from accessing a Topic:-

```
        TopicDefinition topicDef =
        new TopicDefinition(
            TopicDataType.SINGLE_VALUE,
            MetadataFactory.newFieldMetadata(MDataType.STRING));

        topicDef.setProperty(
            TopicProperty.SUBSCRIPTION_HANDLER,

            new TopicSubscriptionHandler() {
                @Override
                public boolean clientSubscriptionRequest(
                TopicClient client,
                Topic topic) throws AuthorisationException {
                    ConnectionType type = client.getConnectionType();
                    if (type.isCategory(ConnectionCategory.MOBILE)) {
                        throw new AuthorisationException(
                            "Mobile clients not permitted");
                    }
                    return true;
                }
            });

        addTopic("NonMobile",topicDef);
```

Delegated Authorisation and Deferred Subscription

Delegating Topic Authroisation Using a Subscription Handler

Another use of subscription handlers is to delegate the authorisation of subscription to some aynschronous process and then performing the subscription on receiving a positive response. The following example shows the use of a subscription handler to delegate the authorisation of a Client's subscription to an Event Publisher and then subscribe when the Publisher gets a response.

```
   @Override
   protected void initialLoad() throws APIException {

        TopicDefinition topicDef =
        new TopicDefinition(
            TopicDataType.SINGLE_VALUE,
            MetadataFactory.newFieldMetadata(MDataType.STRING));

        topicDef.setProperty(
            TopicProperty.SUBSCRIPTION_HANDLER,
            new TopicSubscriptionHandler() {
                @Override
                public boolean clientSubscriptionRequest(
                TopicClient client,
                Topic topic) throws AuthorisationException {
                    try {
                        TopicMessage message =

 theEventPublisher.createDeltaMessage("AuthTopic");
```

Copyright 2013 Push Technology

```
                        message.putFields(
                            client.getClientID(),
                            client.getCredentials().getUsername(),
                            client.getCredentials().getPassword(),
                            topic.getName());
                        theEventPublisher.send(message);
                    }
                    catch (Exception ex) {
                        throw new AuthorisationException(
                            "Failed to send auth request",ex);
                    }
                    return false;
                }
            });
        addTopic("Delegate",topicDef);
    }

    @Override
    protected void messageFromEventPublisher(EventConnection
eventConnection,
    TopicMessage message) {
        if (message.getHeader(0).equals("AuthReply")) {
            try {
                Client client = Publishers.getClient(message.nextField());
                if (client!=null) {
                    Topic topic = getTopic(message.nextField());
                    if (topic!=null) {
                        if (message.nextField().equals("OK")) {

                            topic.subscribe(client);
                        }
                    }
                }
            }
            catch (Exception ex) {
                LOG.error("Error handling subscription reply",ex);
            }
        }
    }
```

The example shows the subscription handler sending a subscription request to an event publisher containing the client id, its credentials and the topic to subscribe. It then returns false so that the subscription does not happen at that point.

When an 'OK' reply is then received from the event publisher the client is subscribed. A negative reply could result in a message being sent to the Client if desired as in this example the Client would not be aware that it had been rejected.

Note that subscription handlers cannot be used with Topics that have Routing Topic Data.

Topic Loading

A Publisher maintains (or provides) one or more Topics, and typically, for each Topic it provides it needs to maintain the state of the data relating to the Topic. Any Client that subscribes to a Topic would therefore need to be provided with the initial state of the Topic's data before it could start receiving updates to the Topic's data.

The act of providing the Client with the current state of the Topic data is called 'Topic Loading'. A special type of Message called a 'Topic Load' Message is provided for this purpose. This allows the Client to distinguish between a load message which represents all of the Topic data and a 'Delta' Message which represents an update to the Topic data.

Subscription Processing

Copyright 2013 Push Technology

When a Client subscribes to a Topic the Publisher is notified via its subscription method. At this point that the Publisher can provide a snapshot of the current Topic state to the Client (a Topic Load).

The default implementation of the subscription method is as follows:-

```
    protected void subscription(Client client,Topic topic,boolean loaded)
    throws APIException {
        if (!loaded) {
            if (topic.hasData()) {
                client.send(topic.getData().getLoadMessage(client));
            }
            else {
                LogWriter logger = getLogger();
                if (logger.isFinestLogging()) {
                    logger.finest("No action in subscription method for
 topic "
                            +topic);
                }
            }
        }
    }
```

This caters for the Topic being pre-loaded by a Topic Loader or the Topic using Topic Data. If there are any Topics which maintain data but do not use Topic Data or Topic Loaders then the subscription method would need to be overridden.

Simple Topic Loading

Simple Topic Loading

The following example shows a Publisher sending a Client a Topic Load Message on subscribing to a Topic:-

```
protected void subscription(Client client,Topic topic,boolean loaded)
throws APIException {
        TopicMessage
  loadMessage = topic.createLoadMessage();
  populateLoadMessage(loadMessage);
  client.send(loadMessage);
}
```

The above example creates a Topic Load Message and populates it with data every time a Client subscribes. It would usually be more efficient to only populate a Topic Load Message when the data actually changes, e.g.

```
private TopicMessage theLoadMessage = null;

protected void initialLoad() throws APIException {
 updateLoadMessage("Initial data");>
}
protected synchronized void subscription(
Client client,Topic topic,boolean loaded)
throws APIException {
    client.send(theLoadMessage);
}
synchronized void updateLoadMessage(String data) throws APIException {
    theLoadMessage=createLoadMessage("MyTopic");
 theLoadMessage.put(data);
```

Copyright 2013 Push Technology

```
}
```

Load Message

The following example shows a Publisher sending Topic Load Message to a Client upon subscription:-

```
protected void subscription(Client client,Topic topic,boolean loaded)
 throws APIException {
     TopicMessage loadMessage = topic.createLoadMessage();
     populateLoadMessage(loadMessage);
     client.send(loadMessage);
 }
```

Cached Load Message

The above example creates a Topic Load Message and populates it with data every time a Client subscribes. It would usually be more efficient to only populate a Topic Load Message when the data actually changes, for example:-

```
private TopicMessage theLoadMessage = null;

protected void initialLoad() throws APIException {
    updateLoadMessage("Initial data");
}

protected synchronized void subscription(
Client client,Topic topic,boolean loaded)
throws APIException {
    client.send(theLoadMessage);
}

synchronized void updateLoadMessage(String data) throws APIException {
    theLoadMessage=createLoadMessage("MyTopic");
    theLoadMessage.put(data);
}
```

In the above example the updateLoadMessage would be called from elsewhere whenever the data changes. The format of the data is not addressed in this simple example.

The above examples also assume that the Publisher has only one Topic which is usually not the case. Where there are multiple Topics then the subscription method would need to return a Topic Load Message for the Topic being subscribed to. This can lead to complex 'if.. else' processing in the subscription method. This can be overcome by using either 'Topic Data Subscription' or 'Topic Loaders'.

Topic Data Subscription

The recommended approach to handling Topic state is using 'Topic Data'. In this case the data for a Topic is held with the Topic. When using this pattern then subscription handling can be as simple as shown below.

```
protected void subscription(Client client,Topic topic,boolean loaded)
throws APIException {
```

Copyright 2013 Push Technology

```
        client.send(topic.getData().getLoadMessage());
    }
```

However, as the default implementation would do this anyway then there would be no need to provide any processing in the subscription method.

For full details of the facilities available for handling data in this way see the section on Topic Data.

Topic Loaders

When multiple Topics are in use (that do not use Topic Data) then it can be inefficient to write code in the subscription method which checks which Topic is being subscribed to and then behave differently. This may be overcome by specifying 'Topic Loaders' for the Topics.

A 'Topic Loader' is an object of type TopicLoader that is declared within a Publisher for a particular Topic (or Topics) using the addTopicLoader method. Whenever a Client subscribes, if a Topic Loader has been defined for the Topic then it is called to perform the Topic Load before the subscription method of the Publisher is called.

Creating a Topic Loader

An example of a Topic Loader might be as follows:-

```
public class MyTopicLoader implements TopicLoader {
    public boolean load(TopicClient client,Topic topic)
    throws APIException {
        TopicMessage loadMessage = topic.createLoadMessage();
        loadMessage.put("Initial Data");
        client.send(loadMessage);
        return true;
    }
}
```

Declaring a Topic Loader

A Topic Loader would normally need to be declared before adding the Topic(s) that it loads. The following example shows a Topic Loader being declared for a single Topic before it is created in the Publishers initial load:-

```
    protected void initialLoad() throws APIException {
        addTopicLoader(new MyTopicLoader(),"MyTopic");
        addTopic("MyTopic");
    }
```

Mixing Simple and Topic Loader Loading

The TopicLoader.load method returns a boolean result to indicate whether it has actually loaded the Topic. This allows a TopicLoader to be employed which only loads some Topics, returning false for those it does not load.

If a Topic has been loaded by a Topic Loader then when the Publisher's subscription method is called the loaded parameter will be true. Therefore if loaded is false it indicates that the subscription method should load the Topic for the Client.

It is therefore possible to use a Topic Loader to load some Topics and allow the subscription method to load others.

Copyright 2013 Push Technology

Topic Loaders for Multiple Topics

To declare a single TopicLoader for multiple Topics it is possible to specify a Topic selector pattern to the `addTopicLoader` method instead of a single Topic name. For example, the following would declare a Topic Loader for the Topic named 'Accounts' and all subordinate to it:-

```
addTopicLoader(new MyTopicLoader(),"Accounts//");
```

Cached Topic Loader

The simple example shown above creates a new Topic Load Message for every subscription. Because this is inefficient the abstract CachedTopicLoader is provided which can be extended to create a tailored Topic Loader which caches the Topic Load message. The abstract class will create the Message and call `populateMessage` to set data in the Message and then send the Message to the Client. However, if called again, rather than recreating the Message it will use the same one as before unless its setDirty method has been called in the interim. This makes it unsuitable for use with multiple topics. It should not be added using a topic selector that matches more then one topic.

The following example shows the use of CachedTopicLoader, declared as an anonymous class. If the `updateData` method is called then the Topic Loader is set as dirty and so on the next Client subscription it will regenerate the Topic Load message by calling populateMessage before sending it to the Client.

```
private CachedTopicLoader theTopicLoader = null;
private Object theData;
private ReentrantLock theLock = new ReentrantLock();

protected void initialLoad() throws APIException {
    theTopicLoader =
        new CachedTopicLoader() {
            protected void populateMessage(TopicMessage message)
            throws APIException {
                theLock.lock();
                try {
                    message.put(theData.toString());
                }
                finally {
                    theLock.unlock();
                }
            }
        };

    addTopicLoader(theTopicLoader,"MyTopic");

    addTopic("MyTopic");
}

void updateData(Object newData) {
    theLock.lock();
    try {
        theData=newData;
        theTopicLoader.setDirty();
    }
    finally {
        theLock.unlock();
    }
}
```

Copyright 2013 Push Technology

Topic Data

Using Topics with special data support or functional capabilities

A Topic typically has data associated with it. A Publisher is responsible for maintaining the state of the Topic's data. This will involve initialising the data as necessary when the Publisher starts and then applying any updates to the data during the time that the Publisher is available.

When a Client subscribes to a Topic then the Publisher should send the Client a Topic Load Message containing the current state of the data. Whenever the data is updated then all Clients that are subscribed to the Topic should be sent only the differences as a Delta Message.

Although what is described above is the normal model for maintaining Topics, Diffusion™ does not impose any strict rules upon how this is achieved or even whether Topics must be used in this way. The data may be maintained in any form and the Messages sent out may be structured in any way. The data may be maintained separately from the Topic or it may be attached to the Topic for ease of processing. The Publisher is responsible for maintaining the integrity of the data at any time and ensuring that any new subscriber gets the latest state followed by any updates required to keep it up to date.

Even though Diffusion™ does not impose data handling mechanisms and Message formats it does provide a feature called "Topic Data" which can greatly simplify the handling of the data associated with a Topic in the following ways:-

- Provides a standard way of encapsulating data in a single TopicData object that can be associated with a Topic. This allows for a simplified 'subscription' process as a standard method can return the current state.
- Provides 'update' mechanisms which automatically compare incoming data with the existing data state and generate deltas that contain only the differences. The generated deltas can then be sent out to all currently subscribed Clients.
- Automatically maintains a cached Topic Load Message which can be returned to new subscribers.
- Allows for Messages to be formatted in various ways, such as Diffusion™ 'Records' format or Google Protocol Buffers. There is also a Custom Data framework to support other data requirements.
- Ensures data integrity by locking data whilst it is being updated. This ensures that no updates can occur to the data whilst a Client is subscribing thus ensuring that the newly subscribed Client receives all subsequent updates.
- Provides a generic metadata modelling feature to allow Message formats to be described programatically.

Where Topic Data provides a data implementation for the Topic it is known as "Publishing Topic Data" but there are also other types of Topic Data that provide functionality on the Topic. For example, features such as 'Paged Topics', 'Service Topics', 'Slave Topics', 'Routing Topics', 'Child List Topics', 'Remote Control Topics' etc.

See the section on "Using Topic Data" for some general information on how to make use of the Topic Data features.

Follow the links below for more information about the various types of Topic Data.

Using Topic Data

Diffusion™ "Topic Data" provides threadsafe management of data associated with a Topic or special Topic functionality.

Various 'types' of data are supported. The type will determine how the Topic handles data (how Messages are interpreted and formatted) or what special feature the Topic will provide. Regardless of the type of data, certain aspects of handling Topic Data are generic. This section describes the generic aspects only. Each type will provide further functionality which will be described in the associated section.

Creating Topic Data

Topic Data is created using the TopicDataFactory class, For example, Topic Data for use with Diffusion™ Record format Messages is created as follows:-

```
RecordTopicData topicData = TopicDataFactory.newRecordData(message);
```

Copyright 2013 Push Technology

Similar methods exist for different types of Topic Data and the TopicData subtype returned will have different capabilities over and above the generic capabilities described here. See the individual sections on each data type for full details.

Associating Topic Data with a Topic

Topic Data is attached to a Topic at the point when the Topic is registered therefore if you are using Topic Data you **cannot** use automatic Topic registration via property file entries. The following example shows Topic Data being added to a Topic during the initialisation of a Publisher:-

```
protected void initialLoad()
throws APIException {
    RecordTopicData topicData =
 TopicDataFactory.newRecordData(MyMetadata.get("Message1"));
    addTopic("MyTopic",topicData);
}
```

In this example it is assumed that a singleton called MyMetadata exists to serve Metadata definitions. This is the recommended pattern rather than defining Metadata within the Publishers. Once a Topic is registered then Clients can subscribe to it and therefore the initial state of the Topic Data must have been set up (if applicable) before registering the Topic.

Subscribing to Topic Data

Topic Data automatically maintains a cached Topic Load Message representing the current state of the Topic.

This greatly simplifies subscription so that all that would be required in the subscription method of a Publisher would be:-

```
protected void subscription(Client client,Topic topic,boolean loaded)
throws APIException {
     client.send(topic.getData().getLoadMessage());
}
```

The above example would only work if all Topics of the Publisher used Topic Data, otherwise it would be necessary to check which Topic was being subscribed to first.

When Topic Data is in use for a Topic then it is also automatically locked against any updates during the time it takes for a Client to subscribe. This ensures that new deltas do not get generated between the time when the Topic Load Message is sent to the Client and the Clients subscription to the Topic is complete.

Message Encoding

It is possible to set the encoding to be used for all load or delta messages generated by the Topic Data.

Lock Timeouts

If a thread tries to lock the data (for example to start an update block) and the data is already locked by another thread then the thread will block until the lock is freed. In order to avoid the thread blocking indefinitely a lock timeout is employed. By default the lock timeout on data is 5 seconds but this can be changed using the setLockTimeout method on the Topic. If a thread cannot acquire a lock within the timeout period then a `com.pushtechnology.diffusion.api.TimeoutException` is thrown.

Publishing Topic Data

"Publishing Topic Data" is the main category of Topic Data which provides threadsafe management of data associated with a Topic, catering for updating data state, deriving differences between inputs and existing data and automatically generating deltas. This includes the special type of SlaveTopicData.

Copyright 2013 Push Technology

Publishing Topic Data Types

There are a number of different types of Publishing Topic Data, each supporting different data formats. The available types are listed below and the related sections describe how to use each type in more detail.

Table 23: Publishing Topic Data Types

Single Value	This supports the simplest form of data where the data for a Topic is defined as a single data item.
Record	This supports a more complex 'Record' format where the data is divided into records and/or fields which are separated by known delimiters within messages. The format of such data is described by 'Metadata'.
Google Protocol Buffers	This supports data defined by the use of Google Protocol Buffers .
Custom Data	This provides an interface such that any data format can be suppoted by writing a data handler.
Slave Topic Data	This is a special form of Publishing data that does not have data itself but instead points to another Topic that does. This allows more than one named Topic to be supported by a single 'master' Topic.

The remainder of this section describes features that are common to all types of Publishing Topic Data.

Initialising Publishing Topic Data

Having created Topic Data it will typically need to have its initial state set up before it is attached to a Topic. The PublishingTopicData interface has an initialise(TopicMessage) method which can be called to initialise from the content of a Topic Message, but different types of Topic Data will have additional initialisation methods. See the various sections relating to the different Topic Data types for full details regarding initialisation.

If the data has not been initialised at the time it is attached to a Topic then it will automatically be initialised to default values as indicated by its Metadata

Updating Publishing Topic Data

Updates to Topic Data may occur in various ways depending upon the nature of the data and its source.

However the updates occur, one of the major features of Publishing Topic Data is that it will automatically compare the inbound data (the update) against the current data state and only generate a Delta Message to send to subscribed Clients if there have been changes.

Also when a Delta Message is generated it will only contain the differences between the current data state and the inbound update. Exactly how this is achieved varies for different types of Topic Data so the relevant sections should be consulted for more detail.

All types of Publishing Topic Data allow an update via the update(TopicMessage) method. This caters for the situation where data updates arrive from other Diffusion™ components such as Event Publishers. The content of the Message is interpreted according to the definition of the data as known to the Topic Data.

According to the type of Topic Data other methods for updating the data may be available. See the relevant sections for more information on update possibilities.

Update Blocks

Topic Data is fully threadsafe. All updates must occur within 'atomic' update blocks where the data is locked, so it is not possible for two threads to update the data at the same time or for a subscription to obtain the data state whilst it is being updated.

An update block is started using the `startUpdate` method and is completed using the endUpdate method.
After updates have occurred then a delta message to send to subscribed Clients can be generated by calling the genetrateDeltaMessage method. It can then be published using a convenience method on the data.

Copyright 2013 Push Technology

The changes to the data are not committed until the `endUpdate` method is called

The following example shows a Publisher `messageFromEventPublisher` method updating its Topic Data:-

```
    protected void messageFromEventPublisher(
    EventConnection eventConnection,
    TopicMessage message) {
        PublishingTopicData topicData =
 getTopic(message).getPublishingData();

        try {
            topicData.startUpdate();
            if (topicData.update(message)) {
                topicData.publishMessage(topicData.generateDeltaMessage());
            }
        }
        catch (APIException ex) {
            LOG.warn("Unable to process messasge from event publisher",ex);
        }
        finally {
            topicData.endUpdate();
        }
    }
```

The try/finally pattern shown above should always be used to ensure that locks are not inadvertently left on the data.

The delta message must be generated within the update block and after all updating is complete. The delta message can be altered (e.g. to set encoding) before publishing but it should be published whilst still within the update block.

The Topic Load message (available via the `getLoadMessage` method) will only be available after the block has successfully ended (If called during the block then a load message representing the state before the block started would be returned).The exception to this would be if the incoming message was itself a topic load in which case the total state of the data would be replaced by the update call. Note that the above example does not handle the possibility of topic loads being received from the event publisher.

Multiple Updates

Any number of updates may occur within a single update block. This is useful for types that allow update at a finer level than the whole message.

It is then possible to check whether any changes were detected before sending a delta message. For example:-

```
    RecordTopicData topicData = getMyData();
        topicData.startUpdate();
        try {
            topicData.update("Record1",0,"Field1","value1");
            topicData.update("Record2",0,"Field2","value2");

            if (topicData.hasChanges()) {
                topicData.publishMessage(topicData.generateDeltaMessage());
            }
        }
        finally {
            topicData.endUpdate();
        }
```

Aborting an Update

If any of the updates within an update block fail (throw an exception) then the update block is automatically ended (locks are released) and the data is reverted to the state that it was in before the block was started. If this occurs then calling endUpdate would have no effect.

Copyright 2013 Push Technology

If other exceptions can occur within the block then they should be caught and the update aborted as shown in the following example:-

```
        RecordTopicData topicData = getMyData();
        topicData.startUpdate();
        try {
            topicData.update("Record1",0,"Field1","value1");

            // Other actions that could throw exceptions
            //

            topicData.update("Record2",0,"Field2","value2");

            if (topicData.hasChanges()) {
                topicData.publishMessage(topicData.generateDeltaMessage());
            }
        }
        catch (Exception ex) {
            topicData.abortUpdate();
        }
        finally {
            topicData.endUpdate();
        }
```

The abortUpdate method may also be used to abort the update for any other reason. Aborting the update releases the locks on the data and ensures that the data is reverted to its state before the update block was started.

Simplified Updating

If multiple updates are not performed and there are no special processing requirements then there is a convenience method on PublishingTopicData which starts an update, updates and publishes a delta (if required).

For example:-

```
topicData.updateAndPublish(message);

// or


topicData.startUpdate();
try {
 if (topicData.update(message)) {
  topicData.publishMessage(topicData.generateDeltaMessage());
 }
}
finally {
 topicData.endUpdate();
}
```

Different types of data may have equivalent convenience methods that take other forms of input.

Simple Updating Example

The following example shows a very simple Publisher with a single Topic using Protocol Buffers Topic Data:-

```
public class SamplePBPublisher extends Publisher {
```

Copyright 2013 Push Technology

```
    private PBTopicData theData;

    protected void initialLoad() throws APIException {
        theData=TopicDataFactory.newPBData("MyMessageClass","MyMessage");
        addTopic("MyTopic",theData);
    }

    protected void messageFromEventPublisher(
    EventConnection eventConnection,
    TopicMessage message) throws APIException {
        theData.updateAndPublish(message);
    }

    protected void subscription(Client client,Topic topic,boolean loaded)
    throws APIException {
        client.send(theData.getLoadMessage());
    }
}
```

This Publisher will receive Messages from an Event Publisher and automatically send out Protocol Buffers Delta Messages when differences are detected. New subscribers will receive a Protocol Buffers Topic Load Message representing the full current state of the Topic.

User Headers

User headers on Messages are not considered part of the state of the Topic. For this reason they are not considered when processing updates (or initialise) from a TopicMessage.

If you want to add user headers to any Topic Load Message returned by the getLoadMessage method then they must first be set using setLoadHeaders.

If you want to add user headers to a Delta Message then you should pass the required headers to the call to generateDeltaMessage.

Acknowledged Messages

If you want Topic Load Messages generated by the Topic Data to be acknowledged then you must indicate this by calling `setLoadAckRequired(true)` before `getLoadMessage`. Note that an acknowledged Topic Load Message will not be cached as it cannot be reused - a new Message will be generated each time.

If you want Delta Messages to be acknowledged then you should use generateAckDeltaMessage

Single Value Topic Data

A single item of data of a specified type (String, Integer, Decimal etc)

Within the Java API Publishing Topic Data can be created which maintains the data related to a Topic as a single data item. This is the simplest form of Topic data that may be associated with a Topic.

Items are formatted strings but special data types (such as INTEGER_STRING and DECIMAL_STRING) allow for more efficient representation and comparison of numbers.

Creating Single Value Data

Single Value Topic Data is created using one of the methods in TopicDataFactory, In the simplest case only the data type needs to be specified. For example:-

```
  theTopicData = TopicDataFactory.newSingleValueData(MDataType.STRING);
```

Copyright 2013 Push Technology

The above method can be used for any type as long as the default values are acceptable. However, if you wish to use a Decimal type with a scale other than the default then you can use:-

```
theTopicData = TopicDataFactory.newSingleValueDecimalData(3);
```

Alternatively you can create a custom data item where the behaviour and format of the item is delegated to a handler as follows:-

```
 theTopicData =
          TopicDataFactory.newSingleValueCustomData(new
DoubleFieldHandler());
```

Whatever type the item is the initial value will be determined by the data type.

Initialising Single Value Topic Data

Single Value Topic Data may be initialised in the generic manner alternatively it is given the default value of the given data type.

In addition, the data item can be initialised before the data is attached to a Topic using a variant of the initialise method which takes any object as a parameter.

For example:-

```
  theTopicData.initialise("123");
```

In the above example a String is supplied but any type can be provided as it will be parsed according to the data type.

Updating Single Value Topic Data

Single Value Topic Data may be updated in the generic manner but in addition there are some simple update methods that are specific to the data type.

```
        theTopicData.startUpdate();
        try {
            if (theTopicData.update("234")) {

theTopicData.publishMessage(theTopicData.generateDeltaMessage());
            }
        }
        finally {
            theTopicData.endUpdate();
        }
```

However, unless special handling of the Message is required then it is much simpler to update and publish a delta in a single call of `updateAndPublish(Object)` which will parse the incoming message and only send a delta if a change is detected.

For example:-

```
theTopicData.updateAndPublish("456");
```

Copyright 2013 Push Technology

Record Topic Data

Data Structured as Records and Fields

Diffusion™ has the concept of Record Data which allows the content of a Message to be represented as String data separated by field and record delimiters.

Within the Java API Publishing Topic Data can be created which maintains the data related to a Topic as one or more Records.

The format of the Topic Data (the Message format) is described using Metadata thus allowing the Topic Data to compare the content of fields within Messages and only generate Delta Messages when there are actual field differences and only populate the fields that have changed. A Record format Topic Load Message is also maintained for serving to new subscribers.

All fields within Records are String format but special data types (such as INTEGER_STRING and DECIMAL_STRING) allow for more efficient representation and comparison of numbers.

The nature of Record data Messages is such that all possible Records and/or fields need to be represented within the Message and therefore fields that have not changed are sent within delta Messages as zero length Strings. This means that it would not be possible for a Client to differentiate between a String field that has not changed and an empty field and therefore it is possible to specify a special character that will be used to represent an empty field.

Record Metadata

Before Record Topic Data can be used it is necessary to describe the Message format in terms of Metadata. See the Records section for more information on how to define Metadata for use with Record Topic Data.

An MMessage created in this way can define one or more Records which make up the Message.

It is recommended that Metadata definitions are encapsulated in a separate class to simplify their use - see Loading Metadata for more information.

Creating Record Topic Data

Record Topic Data is created using the TopicDataFactory, specifying the Metadata that describes the Message format. For example:-

```
RecordTopicData topicData =
    TopicDataFactory.newRecordData(MyMetadata.get("Message1"));
topicData.setEmptyFieldValue(Message.EMPTY_FIELD_STRING);
```

The above example shows use of the recommended value for rendering empty fields. It should be noted that the Clients will need to be able to handle such a value (in this case a single character of value 0x03). If this is not done then the Client would be unable to distinguish between a field that had not changed and one that had been changed to a zero length string. If fields can never be empty then this would not be necessary.

Initialising Record Topic Data

Record Topic Data may be initialised in the generic manner. alternatively it is given the default value of the Metadata.

In addition, the values of individual Records within the data can be initialised before the data is attached to a Topic using a variant of the initialise method which takes a Record (or Records) as a parameter.

```
Record record1 =
 new Record(MyMetadata.get("Message1").getRecord("Record1"));
record1.setField("Name","Gordon Brown");
record1.setField("AccountNumber",99999);
theTopicData.initialise(record1);
```

Copyright 2013 Push Technology

```
Record record2 = new
Record(MyMetadata.get("Message1").getRecord("Record2"));
record2.setField("AddressLine","10 Downing Street","London");
theTopicData.initialise(record2);
```

In the above example the Records within the Message have single multiplicity, however this method may also be used for setting repeating Records by supplying a list of Records. There is also a version of the method which allows a specific occurrence of a repeating Record to be initialised.

Updating Record Topic Data

Record Topic Data may be updated in the generic manner but in addition there are a number of update methods that are specific to the data type.

There are methods that allow updates at a Record level in a similar way to initialise (see above) but individual field values may also be updated.

```
        theTopicData.startUpdate();
        try {
            theTopicData.update(
                "Record1",0,"AccountNumber",12345);
            theTopicData.update(
                "Record2",0,"AddressLine","England");
            if (theTopicData.hasChanges()) {

theTopicData.publishMessage(theTopicData.generateDeltaMessage());
            }
        }
        finally {
            theTopicData.endUpdate();
        }
```

The above example shows single field values being updated but the methods can equally be used to update repeating field values.

When variable repeating Records or Fields are encountered (which can only occur at the end of a Message) if the update has the same number of entries as the current data then they are compared one by one and deltas generated only for changes in the same way as for fixed data. However, when a variable update has a different number of entries from the current data then the whole repeating series will be replaced by those of the update in the delta.

It is also possible to update and publish a delta in a single call of updateAndPublish(TopicMessage) which will parse the incoming message using the associated metadata.

Protocol Buffers Topic Data

Data formatted as Google Protocol Buffers

Diffusion™ supports the use of Google Protocol Buffers as the body of Messages. Before attempting to use Protocol Buffers with Diffusion™ you should read the *Google overview* and understand how to define and use Protocol Buffers generally.

Defining a Protocol Buffer Layout

Unlike Record Topic Data it is not currently possible to define the layout of Protocol Buffers using Diffusion™ generic Metadata. However, Protocol Buffers have their own Metadata in the form of 'proto' files which are compiled using the `protoc` compiler to produce Buffer descriptions that may be used at run time.

When using the Java API and Protocol Buffers Topic Data it is assumed that the protoc compiler has been used to generate Java classes which define the Protocol Buffers that are to be used.

Copyright 2013 Push Technology

The following is an example of a proto file that will be used in examples below:-

```
package testmessages;

option java_package = "com.pushtechnology.diffusion.test.data";
option java_outer_classname = "TestMessagesProto";

enum PhoneType {
  MOBILE = 0;
  HOME = 1;
  WORK = 2;
}

message PhoneNumber {
  required string number = 1;
  optional PhoneType type = 2 [default = HOME];
}

message Name {
  optional string firstname = 1;
  optional string surname = 2;
}

message StreetAddress {
  optional int32 number = 1;
  optional string street = 2;
}

message Address {
  optional StreetAddress streetAddress = 1;
  optional string town = 3;
  optional string state = 4;
}

message Person {
  optional Name name = 1;
  required Address address1 = 2;
  optional Address address2 = 3;
  repeated PhoneNumber phone = 4;
  optional string email = 5;
}
```

Creating Protocol Buffers Topic Data

Protocol Buffers Topic Data is created using the TopicDataFactory, specifying the name of the class generated by `protoc` that contains the Protocol Buffer Message layout and the name of the actual Message within this definition.

For example, to create Topic Data based on the 'Person' definition in the proto example shown above:-

```
       PBTopicData topicData =
            TopicDataFactory.newPBData(
                "com.pushtechnology.diffusion.test.data.TestMessagesProto",
                "Person");
```

The class name must indicate a class generated by `protoc` from a proto definition. This class must be on the callers classpath.

Copyright 2013 Push Technology

The message name refers to a Protocol Buffer Message definition within the class. This is necessary because a proto definition can contain more than one Message definition.

Update Modes

The Topic Data has two modes of operation, "Partial" and "Full" as described below. Partial mode is the default but the mode can be changed after creating the data as follows:-

```
topicData.setUpdateMode(UpdateMode.FULL);
```

Partial Update Mode

In this mode it is assumed that the updates to the data (see below) represent partial updates in that only the optional fields that are to change need be supplied (plus any 'required' items). All supplied fields will be compared to existing field values in the Topic data and a delta message will be generated with only the fields that have changed. Note that the delta will also contain all 'required' fields whether they have changed or not. Also any change to 'repeating' fields (or messages) will result in all occurrences of the repeating field being included in the delta.

Because the absence of an optional field does not indicate its removal in partial mode then to remove optional fields the field needs to be supplied containing the 'deletion value' (see below). Only optional string fields can be removed in this way. The removal of optional fields of types other than string is not supported.

Full Update Mode

In this mode it is assumed that the updates to the data represent the full state of the data. All supplied fields will be compared with the existing field values and a delta message will be generated with only the fields that have changed. Note that the delta will also contain all 'required' fields whether they have changed or not. Also any change to 'repeating' fields (or messages) will result in all occurrences of the repeating field being included in the delta.

The absence of an optional field in the update will indicate that the field is not required and therefore if the base data contained a value for that field then the field will be included in the delta message with a value of the 'deletion value' (see below).

Deletion Notifications

Delta messages that are sent to clients are designed to send only changes to the data in order to minimise the amount of data sent to the client. However, this presents a problem because there is no way within a Googe Protocol Buffer to indicate that an optional field that previously had a value has now been removed. To overcome this the data has the concept of a 'deletion value' which is a string that can be sent as the content of a field to clients in order to indicate that the field has been removed. The deletion value may be a String of any length and a value that would not occur as a valid value in any fields should be chosen. The value may be set as follows after creating the data:-

```
topicData.setDeletionValue("$X$");
```

If no value is explictly specified then a value of "<DEL>" is assumed.

When operating in Partial update mode then this value may also be supplied as the value of a field in the update message in order to indicate the removal of the field.

Deletion notifications only apply to fields of type 'string'. The notification of deletion of all other field types is not supported.

Initialising Protocol Buffers Topic Data

Protocol Buffers Topic Data may be initialised in the generic manner and if not explicitly initialised then it will be initialised to the default values as indicated by the proto definition.

Alternatively the data may be initialised using a variant of the initialise method which takes a Google Protocol Buffers AbstractMessage as a parameter.

Copyright 2013 Push Technology

The following example shows a Google Protocol Buffers Message of type Person being built and used to initialise the data:-

```
Person.Builder person = Person.newBuilder();

person.setName(
 Name.newBuilder().
  setFirstname("Gordon").
  setSurname("Brown"));

person.setAddress1(
 Address.newBuilder().setStreetAddress(
        StreetAddress.newBuilder().
   setNumber(10).
   setStreet("Downing Street")).
         setTown("Westminster").
         setState("London"));

person.addPhone(
    PhoneNumber.newBuilder().
        setNumber("071 2345678").
        setType(PhoneType.HOME));

topicData.initialise(person.build());
```

Updating Protocol Buffers Topic Data

Protocol Buffers Topic Data may be updated in the generic manner but alternatively it can be updated from a Protocol Buffers Message of the same type as the data.

For example:-

```
   void updateData(PBTopicData topicData,Person person) throws APIException
{
       topicData.startUpdate();
       try {
           if (topicData.update(person)) {
               topicData.publishMessage(topicData.generateDeltaMessage());
           }
       }
       finally {
           topicData.endUpdate();
       }
   }
```

It is also possible to update and publish a delta in a single call of `updateAndPublish(TopicMessage)` which will parse the incoming message using the Protocol Buffers message description. Alternatively you could use `updateAndPublish(AbstractMessage)` which is the equivalent to calling `update(AbstractMessage)` to perform the updating.

The data that is input on an update is checked against the current data state and a delta protocol buffer with only those fields that differ will be generated. If there are no differences then no delta would be required. If multiple updates are done within the same block then the update effect will be cumulative in that the the newly generated protocol buffers delta is merged with any existing delta. When repeating fields are used it is important to note that protocol buffers merge repeating fields by appending them. When `generateDeltaMessage` is called then the message will be populated with the cumulative protocol buffers delta.

Copyright 2013 Push Technology

It should be noted that 'required' fields will always be sent in deltas and therefore to minimise message size their use is discouraged. Also, if a 'repeated' field (or message) is changed then it will be necessary to transmit all entries again.

Recommendations for Usage

Because there are some limitations to the use of Google Protocol Buffers then if possible the layout of the Protocol Buffers should observe the following recommendations:-

1. Avoid the use of 'required' fields (or messages) as their values would always need to be transmitted in deltas. In fact Google also recommend that all fields should be 'optional'.
2. Avoid the use of optional field types other than 'string' as the notification of the removal of such fields is not supported.
3. Avoid the use of 'repeated' fields or messages as changes to these would always require the transmission of all field occurrences in the delta.

Using Protocol Buffers within Clients

Within a client application there is no Diffusion™ specific API for handling Protocol Buffers. Instead the user serializes the Protocol Buffer into the Message.

For example, in the Java API:-

```
void writePersonToMessage(Person person,TopicMessage topicMessage)
throws IOException {
     person.build().writeTo(topicMessage.getOutputStream());
}
```

Reading a Protocol Buffer from a Diffusion™ Message is equally straightforward in Java:-

```
 Person readPersonFromMessage(TopicMessage topicMessage)
 throws IOException {
     return Person.parseFrom(topicMessage.getInputStream());
 }
```

When using client applications in other languages then the Protocol Buffers API will be different. See the various Protocol Buffers API implementations for details on each supported language.

Slave Topic Data

An Alias to another Topic

There is a special form of Publishing Topic Data which does not have data itself but instead points to another Topic that does. This allows there to be more than one named Topic supported by the same 'master' Topic.

Slave Topic Data must point to a Topic that has Publishing Topic Data that is not a Slave.

Creating Slave Topic Data

Slave Topic Data creation is shown in the following example:-

```
        RecordTopicData topicData =
            TopicDataFactory.newRecordData(MyMetadata.get("Message1"));

        addTopic("MyTopic",topicData);
```

Copyright 2013 Push Technology

```
        SlaveTopicData slaveData = TopicDataFactory.newSlaveData(topicData);
        addTopic("MySlave",slaveData);
```

This shows a slave topic being defined which points to a master Topic with Record topic Data. In this example any Client that subscribes to MySlave would get the state of MyTopic and any updates to it. Updates could be applied to MySlave and the effect would be to update MyTopic and propagate the changes to any other slaves of MyTopic.

Initialising Slave Topic Data

Slave Topic Data does not need to be initialised as it is the master Topic that contains the data. However, if the initialise method is called on a slave the effect is to call the master's initialise method.

Effects of Updates to the Master Topic

If a Topic has Publishing Topic Data which is pointed to by one or more instances of Slave Topic data then any updates to it are propagated to the slaves. This means that Clients subscribed to the slave Topics will receive deltas of changes to the master Topic.

Effect of Updates to a Slave Topic

It is also permitted for updates to be performed to Slave Topic Data as if it were the actual Topic Data, however all of the type specific update methods would not be available. When a slave is updated then the master Topic is updated and deltas are sent out to Clients subscribed to the master and all slaves.

Efect of Removing the Master Topic

If a master Topic is removed that has slave Topics pointing at it then all slave topics will also be removed.

Custom Topic Data

Data which has user defined formatting or behaviour

Diffusion™ provides the Custom Topic Data API so that it is possible to implement a user written implementation of Publishing Topic Data. This allows for data types and layouts that are not supported directly by Diffusion™.

When Custom Topic Data is used a user written 'data handler' class is nominated which is responsible for maintaining the data, performing comparisons on the data and populating Topic Load and Delta Messages.

Writing a Cutom Topic Data Handler

A Custom Topic Data Handler handles the state of the data for an instance of CustomTopicData.

The handler must implement the `CustomTopicDataHandler` interface. The methods on this interface are called by the Custom Topic Data implementation and should not be called directly. A handler can also extend `AbstractCustomTopicDataHandler` which provides default implementations of optionalm methods.

A handler may initialise the data state in any way that is suitable. The initialisation of the data may occur when the handler is created or as a result of `initialise(TopicMessage)` being called on the Topic Data which in turn will invoke `initialise(TopicMessage)` on the handler.

When the data is associated with a Topic `prepare()` is called to allow for initialisation to be performed (if not done already).

The data may be updated via messages in the normal way via `update(TopicMessage)` on the Topic Data which in turn would call `update(TopicMessage)` on the handler.

Alternatively the data may be updated in some other way, but if this is the case then it is important that it is done within update blocks. If errors occur within updating then this should be reported to the Topic Data using `errorsInUpdate()` on the Topic Data.

When an update block is started `startUpdate()` is called to allow the handler to prepare for updating

Copyright 2013 Push Technology

If the update is aborted then `abortUpdate()` will be called to allow the handler to discard updates.

However the data is updated it is important that the actual state of the data is not updated until `endUpdate()` is called.

The `populateDelta(TopicMessage)` method may be called within an update block to write the delta to the message. This would only be called if `hasChanges()` returns true.

The `populateTopicLoad(TopicMessage)` will be called when a Topic Load is required and should write the current state to the message.

Note that headers or ack flags are set (if required) before the populate methods are called.

Creating Custom Topic Data

Custom Topic Data is created using the TopicDataFactory, specifying the Custom Topic Data Handler to use.

For example:-

```
CustomTopicData topicData =
       TopicDataFactory.newCustomData(
           new MyHandler());
```

Paged Topic Data

Data formatted as 'lines' that a Client can page through

This is a special form of Topic Data that provides the functionality of a Paged Topic.

It is a common requirement for Clients to be able to view data in a 'paged' manner, viewing a page of data comprising one or more lines and then being able to page forwards and backwards through the available data. At the same time this data needs to be managed, adding, updating or removing lines as appropriate. Clients need to be notified of such changes (if it affects them).

Paged Topic Data provides the following functionality to support such a paging paradigm:-

- The ability to manage the data associated with a Topic as 'lines' where there can be any number of lines of data associated with the Topic. Lines can be added, updated or removed.
- The ability to manage the lines in an order specified by a user supplied comparator.
- The ability for each Client subscribing to the Topic to open a view on the data indicating how many lines of data it wishes to see per page.
- The ability for a Client to page through the data, selecting next, prior, first, last or a numbered page.
- Notifications to the client if a change in some way makes its current page invalid. For example, if lines have been added prior to the page making its pagination invalid or if data has changed on the current page. Clients may then refresh the page if required.
- Notifications to the client of updates to lines on its current page.
- Notification to the client of additions to its current page if it is on the last page and has room for more lines.

Table 24: Paged Topic Data Types

String	Where each line is a Java String (i.e. variable length text).
Record	Where each line is a Record comprising a variable number of fields that may be of different data types.

Copyright 2013 Push Technology

Creating Paged Topic Data

String type Paged Topic Data is created as follows:-

```
PagedStringTopicData psData = TopicDataFactory.newPagedStringData();
```

Record type Paged Topic Data (using metadata) is created as follows:-

```
MRecord recordMetadata =
      MetadataFactory.newRecordMetadata("RecMetadata");
  recordMetadata.addField("Currency");
  recordMetadata.addField("Price",MDataType.DECIMAL_STRING);

  PagedRecordTopicData prData =
      TopicDataFactory.newPagedRecordData(recordMetadata);
```

In the above example each line of the data conforms to a simple metadata definition. This is the recommended use but it is possible to specify the metadata as null in which case no metadata rules are applied to the lines of data.

Creating Ordered Paged Topic Data

It is also possible to create Paged Topic Data that will be maintained in an order specified by a Comparator supplied by the user. The Comparator is specified when the Topic Data is created. For example:-

```
  // String type
  PagedStringTopicData psData =
      TopicDataFactory.newPagedStringData(
          new Comparator<String>() {
              @Override
              public int compare(String o1,String o2) {
                  return (o1.compareTo(o2)*-1);
              }
          });
```

The Comparator in the above example would cause the lines to be maitained in reverse alphabetical order.

Note that when using ordered data it is also necessary to indicate what will be done with lines that are evaluated as being equal by the Comparator (i.e. duplicates). The options are as follows:-

Table 25: Duplicates Policies

Policy	Usage
`NOT_ALLOWED`	Duplicates are not allowed. This means that an attempt to add a line that is equal to another line according to the Comparator will cause an exception. This is the default policy.
`FIRST`	If a line is being added that is equal (according to Comparator) one or more existing lines then it will be inserted before exsiting line(s).
`LAST`	If a line is being added that is equal (according to Comparator) one or more existing lines then it will be inserted after exsiting line(s).

So unless the default of `NOT_ALLOWED` is required the policy may be set after creating the Topic Data as follows:-

```
  psData.setDuplicatesPolicy(Duplicates.FIRST);
```

Copyright 2013 Push Technology

Initialising Paged Topic Data

There are no specific methods for intialising the data before adding it to the Topic but if an initial state is required then any of the updating methods described below can be used before the data is added to the Topic.

Updating Paged Topic Data

Paged Topic data can be updated at any time using a set of methods which allow for additions, updates and deletions.

All of these methods will lock the data, perform the update, notify any affected clients of changes as appropriate and then unlock the data. If it is required to lock the data over more than one update so that it cannot be changed whilst manipulating it then the lock() mechanism on the data should be used.

The method signatures vary according to the type so the 'line' referred to in the table below will refer to a String or a Record as appropriate.

Some methods are usable only with unordered Topic Data, some only with ordered and some behave differently according to the type.

Table 26: Usable Methods with Ordered or Unordered Topic Data

`add(line)`	Add the specified line to the end of the data (if unordered) or at the appropriate position (if ordered).
`add(List<line>)`	Add the specified list of lines to the data. For unordered data the lines are all added at the end. For ordered data this is the same as calling add(line) repeatedly.
`add(int,line)`	Add the specified line into the data at the specified index. The line is inserted at the specified index. Indexing starts at 0 so add(0,line) is the same as inserting the line at the start of the data. This may only be used for unordered data - if used on ordered data then an exception will occur.
`add(int,List<line>)`	Add the specified list of lines into the data at the specified index (see above).
`update(int,line)`	Update the line at the specified index with the specified line. This effectively replaces the line with the one supplied. This may not be used for ordered data.
`update(line)`	This may only be used with ordered data. This is functionally equivalent to calling remove(line) followed by add(line).
`remove(int,int)`	Removes one or more lines. The first number specified is the index to start from and the second is the number of lines to remove so remove(0,1) would remove the first line.
`remove(line)`	This may only be used with ordered data. The current line that is equal to the specified line according to the Comparator is removed. If there is more than one matching line then the duplicates policy will specify which would be removed. If no matching lne is found then this has no effect.

There are also methods available to get specified lines or a range of lines which may helpi in updating.

Copyright 2013 Push Technology

Client Notifications on Update

Whenever an update occurs then all Clients affected by the update will be notified as appropriate as described below:-

Additions (at end)

Whenever one or more lines are added to the end of the data then any client that is currently positioned on the last page will be notified of the new lines so that it can add them to its page if desired.

Insertions (additions at index or ordered adds)

Whenever lines are inserted then any client that is positioned on a page that would contain or be after the new lines will be notified that its current pagination is invalid (page is 'dirty'). The client may then choose to mark the page as dirty and refresh it on demand.

Updates

Whenever a line is updated then any client that has the line on its current page will be notified of the new line value so that it can update it if it wishes. Note that an ordered update can reposition the lines and has the effect of a remove followed by an add.

Removals

Removals are treated like insertions. Clients on pages on or after the removal point get a notification that their page is 'dirty'.

Client Handling of Paged Topic Data

How to Handle Paged Topics at the Client

At the client end the client application must be able to handle the Paged Topic Protocol. Most Client APIs will provide the capability for handling such Topics transparently. This section shows how it is handled in the Java Client API.

Handling a Topic Load from a Paged Topic

A Client receives messages on its listener methods and can detect a load message from a paged Topic by means of the isPagedLoad() method. On receipt of such a paged load message the Client application should create a PagedTopicHandler to handle the Topic. This handler will provide the facility to send requests to the Topic and also will route notifications from the Topic to a specified PagedTopicListener. Such a handler can be created using the ExternalClientConnection.createPagedTopicHandler method.

The following code sample shows how to create a suitable Topic handler on receipt of a paged load message:-

```
    public void messageFromServer(
    ServerConnection serverConnection,
    TopicMessage message) {
        if (message.isPagedLoad()) {
            try {

theHandler=theConnection.createPagedTopicHandler(message,this);
            }
            catch (Exception ex) {
                ex.printStackTrace();
            }
        }
    }
```

The returned handler will be of type `PagedTopicHandler` and the above example assumes the calling class implements `PagedTopicListener` and will therefore process notifications.

Copyright 2013 Push Technology

Having created such a handler no further messages will be received for that Topic on the `messageFromServer` method as they will all be consumed by the handler.

Before sending any commands to the Topic or receiving any notifications the Client must 'open' the Topic to establish its view on the Topic data.

Opening a Paged Topic

Having declared a `PagedTopicHandler` for a paged Topic all interactions with the Topic should be via the handler. Before anything else can happen the client application must 'open' the Topic by invoking the `open` method (see Commands below).

When opening, the number of lines per page and the initial page to be returned must be specified.

The number of lines per page can be any number greater than 0 and that will determine the size of the page that will be sent to the client whenever a page is requested. The lines of the data will be paginated for the client according to the number of lines per page requested.

The initial page can be any number greater than 0 for an absolute page number or -1 to indicate the current highest page. If the number specified is greater than the current highest page then the highest page will be returned.

Having opened the Topic the client appplication will then receive the initial requested page on the page method of the `PagedTopicListener` (see Notifications below).

For example, the following would open the Topic for pages with 10 lines per page and the highest page would then be returned:-

```
    theHandler.open(10,-1)
```

Paged Topic Commands

The PagedTopicHandler has the following command methods with which it can send messages to the Paged Topic:-

Table 27: Paged Topic Commands

`open(linesPerPage,initialPage)`	Open the Topic. linesPerPagespecifies the number of lines that the client wants to view per page. `initialPage` specifies the first page to be returned as a page notification (see below).
`page(PageOption)`	Request a page. The page will be returned as a page notification (see below). `PageOption` is an enum with the following possible values:- REFRESH- sends a newly formatted version of the current page. NEXT- returns the next page (or the same page if no more). PRIOR- returns the previous page (or the same page if on first). FIRST- returns the first page. LAST- returns the last (or highest) page.
`page(pageNumber)`	Request a page by absolute page number The page will be returned as a page notification (see below). -1 may be specified to return the highest page. If the page number is higher than the number of pages then the highest page will be returned.
`close()`	Close the Topic. No more notifications will be received after this is called, however the Topic may be opened again at some future point. This does not unsubscribe the Client from the Topic.

The page and close commands may only be invoked after the Topic has been opened.

Copyright 2013 Push Technology

Paged Topic Notifications

The `PagedTopicListener` interface has the following methods upon which it will receive notifications from the Paged Topic:-

Table 28: Paged Topic Notifications

`page()PagedTopicHandler,PageStatus,Lines)`	A page has been returned in response to the open or to a page command. The `PageStatus` will indicate the page number and the total number of pages.The status will never be dirty as this is a fresh page. `TheLines` object will contain all of the lines on the page. There would normally be as many lines as the declared number of lines per page except if it is the last page in which case there may be less.
`statusChanged(PagedTopicHandler,PageStatus)`	The status has changed in relation to the current page that the Client has. This may have happened because the pagination that the client currently knows has changed or it may be because the client should now see the page it has as being dirty because lines have been added or removed before (or on)it.
`add(PagedTopicHandler,PageStatus,Lines)`	One or more lines have been added to the end of the page that the client currently has. This allows the client to add the lines to its view of the page but if it does not do so then it should consider the page to be dirty.
`update(PagedTopicHandler,PageStatus,index,Lines)`	A line on the current page has been updated. The index indicates the relative index of the line within the page (where the first line is 0). The Lines object will contain a single line of data which can be used to replace the line in the client's view. If the client does not replace the line then it should consider the page to be dirty.

Page Status

The PageStatus that is provided with all notifications is an object that contains the following information.

Table 29: Paged Status

CurrentPage	This indicates the current page number. When a page is being supplied as a result of a command then this is the page number as it should be known to the client. When part of a statusChangednotification it is the page number of the page currently known to the client which may have changed due to re-pagination if the page has become dirty.
LastPage	This is the page number of the currently known highest page.
TotalNumberOfLines	This is the current total number of lines available.
Dirty	This will be true if the current page that the client has is now to be seen as out of date. This may be because of the addition or removal of lines on or before the current page. In response to this the client can choose to refresh the page if desired.

Copyright 2013 Push Technology

Lines

The `Lines` object that is passed with some notifications encapsulates the lines of data as either Strings or Records depending upon the type of the Paged Topic Data. There are methods to determine which type the lines are and obtain them as either Strings or Records.

Closing a Paged Topic

Issuing a close command to the Paged Topic via the PagedTopicHandler has the effect of stopping all notifications to the Client. This does not unsubscribe the Client from the Topic and the Topic may therefore be reopened at some future point.

Unsubscribing from a Paged Topic

If a client un-subscribes from a Paged Topic then it should discard any handler in use. If it then wants to re-subscribe to it, it may do so but would need to re-establish the handler as described above.

Service Topic Data

Functional Topic Data that provides an Asynchronous Request/Response Service

This is a special form of Topic Data that provides a request/response service framework.

It is a common requirement for Clients to be able to send some form of command via a Topic to a Publisher which can then perform some processing and optionally return some form of response on the same Topic.

Service Topic Data provides the following functionality to support such a service paradigm:-

- Adding Service Topic Data to a Topic makes that Topic a Service (or command based) Topic where all inbound messages from the Client will be routed directly to the Topic Data instance for execution rather than to the Publisher.
- A user written Service Handler is specified to perform the processing for a Topic and this will be invoked for every command message received on that Topic.
- The handler can return a response immediately (synchronous processing) or return no response but delegate the return of the response to some later process (asynchronous processing).
- Subscription actions (sending of Topic Load) happens automatically, so no handling needed in `Publisher.subscription()` method.
- Automatic timeout mechanism for asynchronous requests.
- Built in error reporting mechanism.

Creating Service Topic Data

Service Topic Data is created as follows:-

```
ServiceTopicData topicData = TopicDataFactory.newServiceData("Service1", new
 MyServiceHandler());
```

Configuring Service Topic Data

Service Topic Data does not have state to be initialised as other Topics do but there are some options to configure the behaviour of the Topic Data before it is added to the Topic:-

Service Data

Even though Service Topic Data does not have data state there is the option to specify some static data for the Topic which will be sent to the Client upon subscription (in the Topic Load Message). The data is specified in the form of a Message which may contain headers and/or data.

The following example shows some service data being specified:-

```
TopicMessage serviceData = createDeltaMessage("Service1");
```

Copyright 2013 Push Technology

```
serviceData.putFields("A","B","C");
topicData.setServiceData(serviceData);
```

Target Topic Name

The `ServiceRequest` object passed to the `ServiceHandler` will contain a Message containing the data from the request passed from the Client. This Message would by default have its Topic name set to that of the Topic that owns the Topic Data. However, this may be changed if required, for example if the request is to be passed to another process that requires a different Topic name. The `setTargetTopicName` method may be used to specify a different Topic name.

When only synchronous processing is in use there would probably be no point in changing the Topic name in the Message.

Header Options

The `ServiceRequest` object passed to the `ServiceHandler` will contain a Message containing the headers from the request passed from the Client but may also have other headers automatically included. This is done using the `setHeaderOptions` method which allows a list of header types to be specified. Any headers specified in this list will be included in the Message within the request before any user headers passed from the Client.

Such headers would only be required for asynchronous processing as all of the specified information is available in the `ServiceRequest` object anyway when processing synchronously. The purpose of adding such headers is only for the sake of a process to which the request Message may be sent.

The following header options are available:-

`SERVICE_TYPE`	The value of the "service type" specified when the Topic Data was created.
`REQUEST_TYPE`	The "request type" is passed with the request from the client. This would be needed to differentiate between different types of requests. However, if there is only ever one type of request then it would not be required.
`CLIENT_ID`	The client identifier of the Client that sent the request.
`REQUEST_ID`	The request identifier sent from the Client with the request which uniquely identifies the request for the Client.

By default no additional headers are added.

Asynchronous Request Timeout

When asynchronously processed the Topic Data will only wait for a certain amount of time before timing out the request and automatically sending a timeout error to the Client. By default this value is set to 5 seconds but may be changed using the `setRequestTimeout` method.

Writing a Service Handler

How to Write the Handler for a Service Topic

The key element of a Service Topic is the user written Service Handler which performs the actual processing. The Service Topic Data itself is merely a framework within which to execute Service Handlers.

Service requests are sent from the Client and routed via the Topic Data to an instance of the Service Handler as specified to the Topic Data.

A Service Handler must implement the `ServiceHandler` interface.

Requests are formatted into `ServiceRequest` objects and passed to the Service Handler on its `serviceRequest` method. The `ServiceRequest` has information like the Client details, a unique request identier (from the Client), a request type and request data in the form of a Message. The request data comes from the

Copyright 2013 Push Technology

Message received from the Client and may comprise user headers and/or data. It is also permissible to have no request data, just a request type.

The Service Handler may process the request and immediately return a `ServiceResponse` object encapsulating the details of the response to send back to the Client. This is ' synchronous ' processing. The Handler may also choose to delegate the processing to some other process which will asynchronously return the response at some later point using the `serviceResponse` method on the Topic Data.

For synchronous processing an error can be reported by throwing a `ServiceException` from the Handler. This is then formatted and returned to the Client. For asynchronous processing a callback to the `serviceError` method on the Topic Data may be used to report a failure in processing.

The following shows an example of a Service Handler that processes a request of type "S" synchronously or delegates a request of type "A" to an event connection:-

```
   private class MyService implements ServiceHandler {
       @Override
       public ServiceResponse serviceRequest(ServiceRequest request)
       throws ServiceException {
           String requestType = request.getRequestType();
           // Sync processing
           if (requestType.equals("S")) {
               try {
                   // Call upon method to process input message and return
an
                   // output message
                   TopicMessage responseMessage =
                       processRequest(request.getRequestMessage());
                   // Returns a response of type "RESP"
                   return new
ServiceResponse(request,"RESP",responseMessage);
               }
               catch (Exception ex) {
                   throw new ServiceException("Service Request Failed",ex);
               }
           }
           // Async Processing
           else if (requestType.equals("A")) {
               EventConnection eventConnection =
                   Publishers.getEventPublisher("MyEventPublisher");
               if (eventConnection==null) {
                   throw new ServiceException("Async service not
available");
               }
               try {
                   if (eventConnection.send(request.getRequestMessage())) {
                       // request successfully submitted to Event Publisher
                       return null;
                   }
                   else {
                       throw new ServiceException("Async service closed");
                   }
               }
               catch (Exception ex) {
                   throw new ServiceException("Async request failed",ex);
               }
           }
           else {
               throw new ServiceException("Invalid Request Type
"+requestType);
           }
```

Copyright 2013 Push Technology

```
        }
    }
```

In the above example asynchronous requests are shown being passed to an Event Publisher. This is only one example of how asynchronous processing can be handled. In this example the response would therefore be returned as a Message from the Event Publisher

The following example shows how such a response may then be returned to the Client via the Topic Data:-

```
    protected void messageFromEventPublisher(
    EventConnection eventConnection,
    TopicMessage message) {
        ServiceTopicData topicData =
            getTopic(message.getTopicName()).getServiceData();
        String clientId = message.getHeader(0);
        String requestId = message.getHeader(1);
        try {
            ServiceResponse response =
                new ServiceResponse(
                    clientId,
                    requestId,
                    "RESP",
                    message);
            topicData.serviceResponse(response);
        }
        catch (Exception ex) {
            topicData.serviceError(
                clientId,
                requestId,
                "Error returning Async response",
                ex);
        }
    }
```

The above example assumes that the response from the Event Publisher has the client and request identifiers in headers. The values of the headers passed via a request Message can be configured for the Topic Data.

Client Handling of Service Topic Data

How a Client handles a Service Topic

At the client end the client application must be able to handle the Service Topic Protocol. Most Client APIs will provide the capability for handling such Topics transparently. This section shows how it is handled in the Java Client API.

Handling a Topic Load

A Client receives messages on its listener methods and can detect a load message from a Service Topic by means of the `isServiceLoad()` method. On receipt of such a load message the Client application should create a `ServiceTopicHandler` to handle the Topic. This handler will provide the facility to send requests to the Topic and also will route responses and errors from the Topic to a specified `ServiceTopicListener`. Such a handler can be created using the `ExternalClientConnection.createServiceTopicHandler` method.

The following code sample shows how to create a suitable Topic handler on receipt of a service load message:-

```
public void messageFromServer(ServerConnection serverConnection,
 TopicMessage message) {
 if (message.isServiceLoad()) {
  try {
```

Copyright 2013 Push Technology

```
    theHandler=theConnection.createServiceTopicHandler(message,this);
   }
   catch (Exception ex) {
    ex.printStackTrace();
   }
  }
 }
```

The returned handler will be of type `ServiceTopicHandler` and the above example assumes the calling class implements `ServiceTopicListener` and will therefore process service responses and errors.

Having created such a handler no further messages will be received for that Topic on the `messageFromServer` method as they will all be consumed by the handler.

Service Type and Service Data

After creating the handler the service type and any service data sent from the server can be obtained from it. The service type identifies the service to the server, and may be used by the client to differentiate between different types of service. The server may also have returned service data which can be used by the service to return any information that might be required by the client. The way in which service type and service data is used is entirely up to the service implementation at the server.

Sending Requests

A service request may be sent to the server via the handler using the request method. A request must specify a request type which must be a request type understood by the service. It may also optionally specify a message containing headers and/or data which will be sent with the request. The message may be used to provide parameters to the service request.

The following example shows the simplest case where no parameters are required to the request:-

```
String requestId = theHandler.request("GetAccounts",null);
```

The method returns a unique request identifier which may be used to correlate the response (or any error) returned from the service with the request.

The following example shows a message being used to pass a parameter-

```
TopicMessage message = theConnection.createDeltaMessage("XYZ",50);
message.put("12435");
String requestId = theHandler.request("GetAccountDetails",message);
```

The Topic name specified for the Message is not important as it will be replaced by the handler in the actual request that is sent.

Handling Responses

Responses to requests are returned via the `serviceResponse` method on the `ServiceTopicListener` specified when creating the Service Handler. The `serviceResponse` method passes a `ServiceTopicResponse` object from which can be obtained the following:-

Table 30: Handling Responses

`requestId`	This is the request identifier that was returned when the request was originally sent.
`responseType`	This is a response type as sent by the service and is used to allow the Client to differentiate between different possible responses.

Copyright 2013 Push Technology

`responseMessage>`	This is a message containing any headers and/or data returned by the service. This can be an empty message if the service did not return any data.

Handling Errors

If the service request fails in any way at the server then an error will be returned via the serviceError method on the `ServiceTopicListener` specified when creating the Service Handler. The `serviceError` method passes a `ServiceTopicError` object from which can be obtained the following:-

Table 31: Handling Errors

`requestId`	This is the request identifier that was returned when the request was originally sent.
`errorType`	This is an enum with one of the following possible values:-
`errorMessage`	This is the error message associated with the error. This will always be present.
`exceptionMessage`	This is an optional exception message which may be returned if the error was due to an exception. This may be null.
`additionalDetails`	This may return optional additional data associated with an exception. This may be null.

Table 32: Error Types

`SERVICE`	An error has occurred whilst executing the service
`INVALID`	The service request was invalid
`TIMEOUT`	The request was executed asynchronously at the server but was timed out before a response was returned.
`USER`	An error was returned by the user written service handler.
`DUPLICATE`	A duplicate request identifier was sent. This cannot happen when using the Java API interface but is present for completeness.

Unsubscribing

When the client application unsubscribes from the Service Topic then the handler will become unusable and any outstanding requests for which responses have not been returned will be discarded.

Routing Topic Data

This is a special form of Topic Data that allows a Topic to point at one or more other Publishing Topics on a Client by Client basis. This means that different Clients may subscribe to the same (routing) Topic but see different data.

The routing (by Client) is determined by a special user written subscription handler that authorises a Client to use the Topic and assigns a real Publishing Topic to represent it. The Client will receive the Topic state (the Topic Load message) from the Topic that it is routed to. Updates to a Publishing Topic linked to in such a way are propagated to Clients routed to it on the routing Topic.

Creating Routing Topic Data

Routing Topic Data is created as follows:-

Copyright 2013 Push Technology

```
RoutingTopicData topicData =
      TopicDataFactory.newRoutingData(new MySubscriptionHandler());
addTopic("MyTopic",topicData);
```

The subscription handler (MySubscriptionHandler) must be an implementation of the RoutingTopicDataSubscriptionHandler interface which provides the Topic routing logic (see below).

The Routing Topic Data Subscription Handler

When Routing Topic Data is created it must have a subscription handler associated with it which is reponsible for providing the Topic routing logic.

The handler should return the actual Topic that the Client should map to or it may return null to indicate the the authorisation and mapping of the Topic is to be delegated to some asynchronous process.

The Client will not actually be subscribed to the routing Topic until the handler returns an actual Topic. In the case where the handler returns null then subscription will not occur until some subsequent process calls the subscribe(TopicClient,Topic) method to provide the mapping.

Synchronous Topic Mapping

The handler can be written to perform any authorisation required and immediately return a Topic to map to.

The following example shows a handler that authorises the subscription using a local class and then returns a different Topic for mobile clients than for all ther types of Client:-

```
private class MySubscriptionHandler
   implements RoutingTopicDataSubscriptionHandler {
       @Override
       public Topic clientSubscriptionRequest(
       TopicClient client,RoutingTopicData topicData)
       throws AuthorisationException {
           if
(MyAuthorisationHandler.canSubscribe(client,topicData.getTopic())) {
               if (client.getConnectionType().isCategory(
                   ConnectionCategory.MOBILE))
               {
                   return theMobileTopic;
               }
               else {
                   return theNormalTopic;
               }
           }
           else {
               throw new AuthorisationException("Not authorised");
           }
       }
   }
```

Asynchronous Topic Mapping

As an alternative mode of operation the subscription handler can delegate the authorisation and mapping of the Topic to some asynchronous process.

The following example shows a handler that delegates the subscription to an Event Publisher and then when a reply is received from the Event Publisher then the mapping and subscription occurs.

The subscription handler shown below sends a message to the Event Publisher with the client id, its credentials and the Topic it is trying to subscribe to. The fact that it returns null means that the subscription will not occur at this point in time.

```
private class MyAsyncSubscriptionHandler
   implements RoutingTopicDataSubscriptionHandler {
```

Copyright 2013 Push Technology

```
        @Override
        public Topic clientSubscriptionRequest(
        TopicClient client,RoutingTopicData topicData)
        throws AuthorisationException {
            EventConnection eventPublisher =
                Publishers.getEventPublisher("Authoriser");
            if (eventPublisher!=null) {
                try {
                    TopicMessage message =
                        Publishers.createDeltaMessage("AuthTopic");
                    message.putFields(
                        client.getClientID(),
                        client.getCredentials().getUsername(),
                        client.getCredentials().getPassword(),
                        topicData.getTopic().getName());
                    eventPublisher.send(message);
                    return null;
                }
                catch (Exception ex) {
                    throw new AuthorisationException(
                        "Failed to send auth request",
                        ex);
                }
            }
            throw new AuthorisationException("Authoriser is not running");
        }
    }
```

The following `messageFromEventPublisher` method would then handle the reply which will indicate if the subscription is OK and return the client id and the Topic that it should map to. This calls back on the Routing Topic Data to do the subscription passing the mapped Topic.

```
    protected void messageFromEventPublisher(
    EventConnection eventConnection,
    TopicMessage message) {
        if (eventConnection.getId().equals("Authoriser")) {
            String result = message.getHeader(0);
            if (result.equals("OK")) {
                String clientID = message.getHeader(1);
                String topicName = message.getHeader(2);
                Client client = Publishers.getClient(clientID);
                if (client!=null) {
                    Topic topic = getTopic(topicName);
                    if (topic!=null) {
                        RoutingTopicData topicData =
                            (RoutingTopicData)getTopic("MyTopic").getData();
                        try {
                            topicData.subscribe(client,topic);
                        }
                        catch (Exception ex) {
                            LOG.warn("Subscription failed",ex);
                        }
                    }
                }
            }
        }
    }
```

Client Subscription

When a Client subscribes to a Routing Topic then the subscription request is intercepted and passed to the Routing Topic's subscription handler (see above). The handler may return a Topic to map to in which case the Client will

Copyright 2013 Push Technology

immediately become subscribed to the Routing Topic. Alternatively the handler may delegate the authorisation/mapping in which case the Client will not be subscribed but can be subscribe later by a callback on the Routing Topic Data.

On subscription the `Publisher.subscription` method will be called and the Topic Load message can be obtained from the routing data, but it must be obtained using the `getLoadMessage(client)` method rather than the no arguments variant as the load message will vary per client. The default implementation does this anyway.

Messages Published To Clients

Whenever a message is published on a Topic that is mapped to by a Routing Topic then a message will also be published to all Clients that map to that Topic but the Topic name in the message will have been altered accordingly.

Messages Sent from Clients

Any messages sent from Clients on a Routing Topic are routed to the Publisher in the normal way.

Effect of Removal of a Target Topic

If a Topic is removed that has been mapped to by a Routing Topic then all Clients that have been mapped to that Topic will be unsubscribed from the Routing Topic.

Remote Topic Data

There are special forms of Topic Data that support the Remote Control framework.

For a full description of how to use Remote Control see the Remote Control section. This section describes how to set up the Topic Data components in order to enable it within a Diffusion™ Server.

Remote Control Topic Data has two forms: that created at the 'Edge' tier and that created at the 'Relay' tier.

Edge data is used for both two and three tier architectures whereas Relay data is used only in three tier architectures.

Edge tier data is deployed in a Publisher which interfaces directly with Clients.

Relay tier data is deployed in a Publisher in an intermediate Server between the Edge and the back end Remote services. The Relay layer is used to isolate the back end services from the front end Client processing.

Creating Edge Remote Control Topic Data

Remote Control Topic Data (type `RemoteControlTopicDataEdge`) which is to be used at the Edge for a simple two tier architecture is created as follows:-

```
RemoteControlTopicDataEdge topicData =
 TopicDataFactory.newRemoteControlData();
```

A Remote Control Topic may then be created using such Topic data and this Topic will then provide a connection point for applications using the Remote API. An application using the Remote API which wishes to register a remote service must specify the name of the Remote Control Topic.

Configuring Edge Remote Control Data for Three Tier Processing

Remote Control Topic data which is to be used at the Edge in a three tier architecture is created as follows:-

```
ServerDetails serverDetails =
    ConnectionFactory.createServerDetails(
  "dpt://"+RELAY_HOST+":"+RELAY_PORT);

serverDetails.setInputBufferSize(256*1024);

RemoteControlTopicDataEdge topicData =
 TopicDataFactory.newRemoteControlData();
```

Copyright 2013 Push Technology

```
topicData.setRelay(new RemoteControlRelayDetails(serverDetails));
```

A Remote Control Topic may then be created using such Topic data and this will attempt to connect (using a Publisher Client connection) to a Relay Remote Topic with the same name at the specified Server and port. The creation of the Topic Data for the Relay tier is described below.

In the above example the Server Details that are set up specify the location of the Server hosting the Publisher that deploys the Relay Topic Data. The setting of the input and output buffer sizes is important to performance and should match the buffer sizes used at the corresponding connector of the relay server.

The setting of the outbound queue definition is also important to performance. This must specify the name of a queue definition in the *etc/Server.xml* properties which has sufficient capacity to cater for outbound messages to the relay. It should be noted that the connector at the relay server should also have an adequate queue definition.

If the Relay connection cannot be made as soon as the Topic is created then it will keep retrying until a connection is established therefore it is not essential that the Relay Server is started before the Edge Server.

It is essential that the Control Topic at the Relay Server has the same name as the one created at the Edge.

In a three tier architecture Remote Service applications do not connect directly to the Edge tier but connect via the Relay tier (see below).

Creating Relay Remote Control Topic Data

Remote Control Topic Data that is to be used as a 'Relay' component within a three tier architecture is created as follows:-

```
RemoteControlTopicDataRelay topicData =
 TopicDataFactory.newRemoteControlRelayData();
```

As soon as the Topic is created then a Control Topic at the Edge will be able to connect to it and applications using the Remote API will be able to connect and register services with it.

Remote Service Topic Data

When a Remote application registers a service with a Remote Control topic then a 'Service Topic' will be automatically created which has RemoteServiceTopicData. This form of Topic Data cannot be explicitly created.

Child List Topic Data

Functional Data that Maintains a List of Child Topics and Notifies Changes

Within the Java API Topic Data can be created which automatically maintains a list of the child Topics of the owning Topic and sends out delta messages when a child is added or removed.

Creating Child List Topic Data

Child List Topic Data is created as follows:-

```
theTopicData = TopicDataFactory.newChildListData();
```

Using Child List Topic Data

Child list Topic Data does not need to be initialised.

The data is added to the Topic when the Topic is added in the usual manner.

Copyright 2013 Push Technology

The topic load message is automatically maintained and sent out to any client that subscribes to the Topic. There is no need to implement anything in the Publisher's subscription method to do this (the loaded flag passed to this method will be 'true').

Child list Topic needs no updating as it is all maintained automatically.

When a child topic is added or removed then a delta message is automatically sent to all clients subscribed to the Topic. The delta message has a single header with value "A" (added) or "R" (removed) and the data body contains the node name of the child Topic.

Topic Notify Topic Data

Functional Topic Data that Allows Notification of Topic Creations to be Sent to Clients

This is a special form of Topic Data that allows Clients to be notified when a new Topic is added. This is especially useful for Publisher Clients which may want to replicate Topics from the Master server.

A Client can select which part of the Topic tree it wants to receive notifications for and also the level of notification it requires. For example, it can get only notification of Topic name, or it can also request the Topic Metadata (where there is some) or it can even request the full definition of the Topic which would allow the Topic to be replicated (useful mainly in Publisher Clients).

Creating Notify Topic Data

Topic Notify Topic Data is created as follows:-

```
TopicNotifyTopicData topicData =
    TopicDataFactory.newTopicNotifyData(false);

addTopic("TopicNotifier",topicData);
```

The boolean parameter that is specified on creation indicates whether to use metadata caching. This can minimise the amount of Metadata that is sent to a Client by only sending a named Metadata defintion once and thereafter sending only its name. However, this will only work if the root names of all Metadata used in all Topic definitions are unique as it is the top level name that is used to notify clients. If the uniqueness of metadata names cannot be guaranteed then you should not use metadata caching as unpredictable results could occur.

Using Notify Topic Data

From the Server/Publisher point of view once a Topic Notify Topic is created there is no further processing required. Once the Topic is available Clients may subscribe to it and request Notifications.

Client Handling of Notify Topic Data

How a Client handles a Topic Notify Topic

At the client end the client application must be able to handle the Notify Topic Protocol. Some Client APIs will provide the capability for handling such Topics transparently. This section shows how it is handled in the Java Client API.

Handling a Topic Load

A Client receives messages on its listener methods and can detect a load message from a Topic Notify Topic by means of the `isTopicNotifyLoad()` method. On receipt of such a load message the Client application should create a `TopicNotifyTopicHandler` to handle the Topic. This handler will provide the facility to select the range and level of notifications and also will route notifications to a specified `TopicNotifyTopicListener`. Such a handler can be created using the `ServerConnection.createTopicNotifyTopicHandler` method.

Copyright 2013 Push Technology

The following code sample shows how to create a suitable Topic handler on receipt of a Topic Notify load message and set it to received full Topic add notifications:-

```
    public void messageFromServer(
    ServerConnection serverConnection,
    TopicMessage message) {
        if (message.isTopicNotifyLoad()) {
            try {
                theHandler=theConnection.createTopicNotifyTopicHandler(
                    message,
                    this);

 theHandler.setNotificationDetails(NotificationLevel.FULL,false,false);
            }
            catch (Exception ex) {
                ex.printStackTrace();
            }
        }
    }
```

The returned handler will be of type `TopicNotifyTopicHandler` and the above example assumes the calling class implements `TopicNotifyTopicListener` and will therefore process notifications of Topics added.

Having created such a handler no further messages will be received for that Topic on the `messageFromServer` method as they will all be consumed by the handler.

It is possible to request notifications of any of the following:-

- Topic Additions - Notification of a Topic being created.
- Topic Deletions - Notifications of a Topic being removed.
- Topic Updates - Notificatons of changes to non static Topic Properties. At this point in time only the Topic reference is catered for.

Add Notification Levels

When a handler is created the required Notification Level for Topic add notifications should be specified. The level indicates the amount of information that is notified to the Client for each (selected) Topic that is added. The amount of information should normally be set to the least required. Full infirmaton would normally only be required when replicating Topics (in a Publisher Client).

The possible Notification Levels are as follows:-

Table 33: Notification Levels

Level	Description
MINIMUM	Only the Topic name and type are notified. This is the default if not explictly set.
PROPERTIES	The Topic name, type and all Topic properties that are set are notified. The properties are all of the configurable settings of the Topic other than its metadata.
METADATA	The Topic name, type and Metadata (for Topic types that have metadata) is notified.
FULL	All details are notified - name, type, properties and metadata.
NONE	This can be used to indicate that add notifications are not required. This may be the case if the client only wishes to receive delet and/or update notifications.

The Notification level can be changed at any point whilst the Client is subscribed to the Topic (using the `setNotificationDetails` method on the handler) but it is recommended that the level is chosen at the beginning an retained for the duration of the subscription.

Copyright 2013 Push Technology

Selection of Topics to Notify

Creation of the handler will not cause the Client to start receiving any notifications of Topics added. The range of Topics for which notifications are required must be set using the `select` method on the handler. The range is specified as a set of Topic Names and/or Topic Selector patterns. Once the selection has been received by the server then if any Topics that exist at the Server match the selections then the Client will be notified immediately and if any Topics are created that match the selections in the future (whilst the client is subscribed to the notifying Topic) then the Client will be notified at the time of creation.

Notifications of new Topics would always occur before the Client receives a Topic Load from newly added Topic.

The set of selections is specified as a TopicSet and each time select is called it can be used to add to or update the current selection at the server. Selection modes for the specified set are:-

Table 34: Selection Modes

Mode	Description
ADD	The set of Topic selectors are added to the set of selections at the server.
REPLACE	The set of Topic selectors will totally replace the set of selections at the server.
REMOVE	The set of Topic selectors will be removed from selections at the server. Onl if the selecto pattern is an exact match for a previously supplied pattern will it be removed.
CLEAR	Removes all selections at the server.

It should be noted that this is an asynchronous operation with the server and notifications relating to the previous selections may continue to be received by the Client until the update is successfully processed by the server.

For example, the following code could be used to select a set of notifications:-

```
theHandler.select(SelectionMode.ADD,new TopicSet("Accounts//","Trading//"));
```

The above would cause notificatons for all Topics created under the root level "Accounts" and "Trading" Topics.

Handling Notifications

All notifications of Topics added will be passed to the `TopicNotifyTopicListener` object specified when the `TopicNotifyTopicHandler` was created.

Topic Adds

Notifications of Topic added are passed on the `topicAdded` method which has two parameters as follows:-

`String topicName`	The full name of the Topic that has been added
`TopicDefinition definition`	The encapsulated details of the Topic. Depending upon the notification level selected some details may not be available. For example if metadata notification was not selected then the getMetadata method would return null. The Topic type available as a `TopicDataType` - see javadoc of this for full details of all of the Topic types that can occur. The Topic properties are provided as a Map of key value pairs. The key will be an instance of `TopicProperty` - see the Javadoc of this for full details of all of the properties that may be returned. If properties are not selected the returned Map would be empty. The Topic metadata is returned as an instance of MNode where appropriate. If the Topic Type does not have metadata or metadata notifications were not selected then this would be null.

Copyright 2013 Push Technology

The following example of a `topicAdded` implementation shows type, metadata and a single property being extracted from the definition passed:-

```
public void topicAdded(String topicName,TopicDefinition definition) {
 TopicDataType type = definition.getType();
 MNode metadata = definition.getMetadata();
 try {
  int capacity =
 definition.getIntegerProperty(TopicProperty.DELTA_MESSAGE_CAPACITY);
 }
 catch (APIException ex) {
  ex.printStackTrace();
 }
}
```

Topic Removals

Notifications of Topic removals are passed on the `topicRemoved` method which simply passes the Topic name.

Topic Updates

Notifications of Topic updates are passed on the `topicUpdated` method which passes the Topic name and a map of properties that have changed. At this point in time the only property that is catered for is the Topic reference (keyed by `TopicProperty.REFERENCE`).

Unsubscribing

When the client application unsubscribes from the Topic Notify Topic then the handler will become unusable and any outstanding notifications will be discarded.

Metadata

Metadata is 'data that describes data' and in Diffusion™ terms it is something that defines the format of a Diffusion™ Topic Message.

Diffusion™ Metadata is a generic mechanism for describing Message formats regardless of the exact data representation. It describes a Message in terms of 'Fields' within it and these fields may be grouped into 'Records' which themselves may be further subdivided into Fields and/or Records.

This generic representation of Metadata can potentially be used for many different types of Message data. For example, Metadata may be used to describe a Diffusion™ Record based Message where Fields and Records have the same meaning but certain constraints exist (e.g. nesting of Records is not permitted).

The purpose of Metadata is to allow for programatical modelling of data structures. At this point in time it is only available within the Java API and used in the following ways:-

- To define the layout of a Record so that fields within the Record can be addressed by name.
- To define the layout of the Messages used by Record Topic Data in terms of one or more Record definitions.

Metadata Structure

A Metadata definition is made up of a hierarchy of Metadata 'nodes' (class `MNode`) with a Message node (class `MMessage`) at the top. A Message is defined as one or more Field (class `MField`) and/or Record(class `MRecord`) nodes. A Record is also defined as one or more Field and/or Record nodes (thus `MMessage` is a specialisation of `MRecord`). A Field defines an elementary data item of a particular data type.

Every node has a name which must be unique within its parent node. Every child node may represent one or more possible occurrences of that Record or Field within the parent. The number of possible occurrences of a node is described by its multiplicity.

The order in which Nodes are defined within their parent defines the order that they will appear in within a Message.

Copyright 2013 Push Technology

Message Metadata

A Metadata Message (class `MMessage`) is the top level object of a Metadata definition which represents a whole Message layout. Metadata Messages are created using the `MetadataFactory` class specifying the type of Topic Data that the Metadata describes. Even though Metadata definitions are generic in form, the type of data may impose constraints on the Metadata. A Message may be made up of one or more Fields and/or Records.

Record Metadata

A Metadata Record (class `MRecord`) defines a group of Fields and/or Records within a parent Message or Record. A Record has a multiplicity within its parent node.

Field Metadata

A Metadata Field (class `MField`) defines an elementary data item within a Message or Record. Every Field has a data type which defines its actual representation within the Message. A Field may have a default value specified to use when initialising Message data. A Field has a multiplicity within its parent node.

Data Types

The data type of a Field defines its actual representation within the Message. The data types available will vary according to the Topic Data Type that the Metadata is defining. The same data type may result in different physical data representations for different Topic Data Types. The available data types are defined by the enum `MDataType` and the following are currently available:-

Table 35: Data types

STRING	A character string. Represented internally as a `java.lang.String` . The initial default value for this type is "" (a zero length String).
INTEGER STRING	An integral number represented in the Message as a character string. This is represented internally as a `java.math.BigInteger` . If a Field is defined as this type then it can only contain numeric digits with an optional leading sign. By default, empty fields are not allowed. The initial default value for this type is "0".
DECIMAL STRING	A decimal number represented in the Message as a character string. This is represented internally as a `java.math.BigDecimal` . Decimal fields have the number of places to the right of the decimal point defined by the ' scale', the default being 2. Such values may be parsed from a character String with any number of digits to the right of the decimal point but 'half up' rounding will be applied to achieve the target scale and output of the field will be rendered with the specified scale. By default, empty fields are not allowed. The initial default value for this type is "0.00" (depending on scale). For comparison purposes the scale is ignored therefore a value of 1.50 is the same as 1.5.
CUSTOM STRING	This is a special type where the behaviour is delegated to a user written `CustomFieldHandler` . See Javadoc for more detail or the example below. This type is available in all Topic Data Types.

`CustomFieldHandler` Example

A Custom Field Handler can be used for defining a data type other than those provided and can deal with any String data format required. The following example shows a handler used to represent a Double value:-

```
public class DoubleFieldHandler implements CustomFieldHandler {

    public Object getInitialDefaultValue() {
        return new Double(0.0);
```

Copyright 2013 Push Technology

```
    }

    public Object parse(Object object) throws APIException {
        if (object==null) {
            return new Double(0.0);
        }
        try {
            return (Double.parseDouble(object.toString()));
        }
        catch (Throwable ex) {
            throw new APIException(
                "Unable to parse "+object+" as double value",
                ex);
        }
    }

    public boolean areEqual(Object source,Object target) {
        return source.equals(target);
    }
}
```

Scales

Decimal format Fields (such as `DECIMAL_STRING)` may have a scale which defines the number of places to the right of the decimal point.

The scale of a Field is set using the `setScale` method. If not specified then 2 is assumed. This value is ignored for all but decimal format fields.

Default Values

Every field may have a default value specified for it using the `setDefaultValue` method. If not specified then the default value is assumed to be the default for the data type. When a Message is created using Metadata then default initialisation will apply the default values specified for each Field.

Empty Values

By default `STRING` type fields will accept an empty field (i.e. a zero length string) as input but other types do not. However it is possible to allow other field types to allow empty input also. This is done using the `setAllowsEmpty` method on the `MField` object.

Multiplicity

The 'multiplicity' of a Metadata Field or Record defines the number of times the corresponding data can occur within its parent.

Multiplicity (defined by the `Multiplicity` class) is defined in terms of the minimum and maximum number of occurrences. Some data representations (such as Protocol Buffers) support variable numbers of nodes, whereas others (such a Record Data) only support fixed number of nodes (where minimum=maximum) except in the last position.

Fixed multiplicity can therefore be defined by a single number. For example a multplicity of 5 (e.g. `new Multiplicity(5)`) indicates that there must be exactly 5 occurrences of the node within its parent.

Variable multiplicity is defined in terms of a minimum value and a maximum value and is represented with the notation `n..n` . So a multiplicity of `1..5` (`new Multiplicity(1,5)`) indicates that there can be between 1 and 5 occurrences of the node within its parent. A special maximum value of -1 is used to represent no maximum so a multiplicity of `1..n` (`new Multiplicity(1,-1)`) means that there can be any number of occurrences of the node but there must be at least one.

Optional nodes are indicated by a a minimum value of 0. So `0..1` would represent a node that can occur 0 or once and `0..n` would represent a node that can occur any number of times or not at all. Of course a fixed multiplicity of 0 is not allowed.

Copyright 2013 Push Technology

Record Data Example

The example below shows a method that defines some Message Metadata to be used with Record Data .

The Message comprises two Records.

```
MMessage defineMyMetadata() throws APIException {
        MMessage message =
            MetadataFactory.newMetadata("MyMessage",TopicDataType.RECORD);

        MRecord record=message.addRecord("Record1");
        MField field;
        field=record.addField("Name");
        field=record.addField("AccountNumber",MDataType.INTEGER_STRING);
        field.setDefaultValue(-1);
        field=record.addField(
            "Price",
            MDataType.DECIMAL_STRING,
            new Multiplicity(2));
        field.setScale(3);

        record=message.addRecord("Record2");
        record.addField("AddressLine",new Multiplicity(5));

        return message;
    }
```

Loading Metadata

Currently Metadata can only be described programatically so it may be useful to employ the singleton pattern to encapsulate Metadata definitions so that they only need to be loaded once. The following class is an example of such a singleton:-

```
public class MyMetadata {

    private static MyMetadata theInstance = null;

    private HashMap<String,MMessage> theMetadata =
        new HashMap<String,MMessage>();

    private MyMetadata() throws APIException {
        loadMetadata();
    }

    private static MyMetadata instance() throws APIException {
       if (theInstance==null) {
            synchronized(MyMetadata.class) {
                if (theInstance==null) {
                    theInstance=new MyMetadata();
                }
            }
        }
        return theInstance;
    }

    public static MMessage get(String name) throws APIException {
        return instance().getMetadata(name);
    }

    private MMessage getMetadata(String name) {
```

Copyright 2013 Push Technology

```
        return theMetadata.get(name);
    }

    private void loadMetadata() throws APIException {
        loadMessage1();
        loadMessage2();
        //... etc
    }

    private void loadMessage1() throws APIException {
        MMessage message =
            MetadataFactory.newMetadata("Message1",TopicDataType.RECORD);

        MRecord record=message.addRecord("Record1");
        record.addField("Name");
        record.addField("AccountNumber",MDataType.INTEGER_STRING);

        record=message.addRecord("Record2");
        record.addField("AddressLine",new Multiplicity(5));

        theMetadata.put("Message1",message);
    }

    private void loadMessage2() throws APIException {
        // load another message format
    }
}
```

Metadata definitions could then be obtained using the MyMetadata.get(name) method, for example:-

```
    MMessage metadata = MyMetadata.get("Message1");
```

Topic Selection

There are a number of places throughout Diffusion™ where Topic names need to be specified. For example, by Clients when subscribing. In order to simplify the processing required, particularly when dealing with hierarchic Topics, there are many places where Topics may also be specified using 'Topic Selectors'.

Topic Selectors

A Topic Selector is a string which may be used to select more than one Topic in the following ways:-

- By indicating that subordinate Topics are to be included.
- By 'fuzzy matching' on Topic Names.

Including Subordinate Topics

When specifying a Topic name you can also indicate that all of its subordinate Topics are to be included by suffixing the name with a '/'.

For example to select all of the subordinate Topics of a Topic named "MyTopic" you would use a selector of the format:-

```
MyTopic/
```

This notation may also be used with hierarchic Topic names. So to select all Topics subordinate to the Topic named "A/B" you would use a selector of the format:-

Copyright 2013 Push Technology

```
A/B/
```

Specifying a suffix of '/' does not include the Topic named prior to the final '/'. In order to include the specified Topic and all of its subordinates then "//" should be used.

For example, to select the Topic "A/B" and all of its subordinates you would specify:-

```
A/B//
```

Fuzzy Matching

A number of Topics can be selected using a single Topic Selector which uses 'regular expressions' (commonly known as 'regex') to match against Topic (node) Names.

Regular expressions provide a powerful mechanism for String pattern matching which will not be discussed here. A Java Tutorial is available for those familiar with Java or a more generic tutorial can be consulted for other language users. The regular expression syntax supported by Diffusion™ is defined by the Java Pattern class.

The important point to note about the use of regular expressions in Topic Selectors is that they are hierarchic (separated by /) and that each node of a selector can be a regular expression pattern. A multi-node selector is evaluated node by node starting from the Topic tree root in an attempt to find a matching Topic in the tree.

If there is no "/" in the selector string but there are regex characters then the whole string is applied as a regex against the full topic name. This can be useful in some circumstances, for example when you do not know how many topic levels exist but it is far less efficient than node level matching.

Example of Selector Regex Processing

Consider the following Topic Selector:-

```
A.*/B/.*X
```

As the above selector has 3 node specifications it will only match with Topic names with 3 parts.

The first check will be to select any Topics directly under the Topic Tree root whose name matches the pattern 'A.*'. In regex notation this pattern will match with any String that starts with 'A' and is followed by zero or more other characters. So if there are no top level topics that start with an 'A' then this selector would match with no topics at all. However, for any top level Topic that does match processing will go to the next node which is a simple String 'B' and would only match with a subordinate Topic called 'B'. So if any Topics called 'B' are found under the top level Topics starting with 'A' then processing goes onto the final pattern '.*X' which in regex notation indicates any String with any number of characters before a final 'X' (a String ending with 'X').

So to summarise, the above selector will match only with hierarchical Topic names with 3 elements where the first element starts with an 'A', the second element is 'B' and the third element ends with an 'X'.

The following are therefore matches:-

```
Accounts/B/TAX
```

```
A/B/X
```

```
Admin/B/ProjectX
```

```
But the following are not:-
```

```
Accounts/TAX
```

```
Admin/B/ProjectYHR/B/TAX
```

If you do not know how many levels you are dealing with but want to select topics where the lowest level Topic name is "ABC" then you would need to use a whole Topic name selector as follows:-

```
'.*\x2FABC'
```

The use of '\x2F' is necessary to represent a '/' as otherwise the selector would evaulate per level.

So this selector would match with:-

```
A/B/ABCX/Y/ZZZ/wwwwww/ABC
```

Copyright 2013 Push Technology

but not with:-

```
A/B/CABC
```

Mixed Mode Selectors

It is permitted to mix regex pattern handling and subordinate Topic suffixes in the same Topic Selector pattern.

So it is permitted to use a selector of the form "A.*/Address/" which would have the effect of selecting all of the Topics subordinate to the Topic named 'Address' within any top level Topic starting with 'A'.

Selector Examples

Table 36: Selector Examples

Selector Pattern	Selects	Examples
A/	All Topics subordinate to, but not including the top level Topic named 'A'.	A/B, A/B/C
A//	The top level Topic named 'A' and all Topics subordinate to it.	A, A/B, A/B/C
A.*	All Topics that start with the letter 'A'	A, Accounts, Admin
A.*//	All Topics that start with the letter 'A" and their subordinates.	A, Accounts, A/B, Admin/X/Y
Counties/.*ex	All Topics under the Top level Topic called 'Counties' whose name ends in ' ex '.	Counties/Middlesex, Counties/Sussex
.*folk	All Topics whose full name ends in ' folk ' regardless of level.	UK/Counties/Suffolk, Counties/Norfolk

When is a Topic String a Selector?

In certain parts of the APIs only Topic names may be specified (for example when adding a Topic) but in other areas Selectors are allowed (for example in subscription).

A Topic String is considered to be a selector if it is terminated by a '/' and/or it contains any one of the regex metacharacters which are defined as:-

```
[]\^$.|?*+()
```

Topic Fetch

It is possible for a Client to fetch the current state (represented as a Topic Load message) at any point, even if the Client is not subcribed to the Topic. It is even possible to fetch the state of a Topic that does not exist (allowing for a simple request/response mechanism).

This feature will work out of the box for Topics that use Topic Data but for other Topics there is a need to provide the handling of a fetch in the Publisher.

Client Fetch

How to fetch the state of a Topic from a Client

The Client would request this using a 'fetch' command on the API.

Copyright 2013 Push Technology

For example in the Java API this is done using the `fetch` method which can be used for a single Topic or for a set of Topics.

Topic Selectors may be specified to the fetch request in which case a fetch will be performed for all matching Topics.

It is also possible to provide one or more headers with the fetch request which will be reflected back in the reply for correlation purposes. If the reply message already has user headers then the request headers would be appended to the end of the header list.

The fetch request is an asynchronous request and the reply (the Topic state) will be returned to the Client on its normal listener interface. In the Java client this is on the usual `messageFromServer` method and messages that are fetch replies can be distinguished by means of the `isFetchReply` method on the incoming message.

Server Fetch Handling

What happens at the Server when a fetch request is received from a Client

When a request to fetch an existing Topic is received by the Server then one of the following will be done to obtain the Topic state.

- The Client Authorisation Handler will be called to determined whether the Client can fetch the Topic state. There are separate methods on this handler for dealing with normal Topic names and Topic selectors. If the handler rejects the fetch then no further action will be taken.
- If a "Fetch Handler" (see below) exists for a Topic then it will be called to obtain the Topic state. The handler may return null to indicate that the fetch is not allowed or that it is to be handled asynchronously.
- If the Topic has Topic Data (and there was no fetch handler) then the current state will be obtained from the Topic Data.
- If there was no fetch handler and no Topic Data then as a last resort the fetch is delegated to the `fetchForClient` method in the Publisher.

Fetch for Non Existing Topics

When a request to fetch the state of a Topic that does not exist is received then it is reported as an event via the `ClientListener` interface on its `clientFetchInvalid` method. This routine could then send a reply to the Client via the `Client.sendFetchReply` method thus allowing a fetch to even be used for a Topic that does not exist.

Fetch Batching

As a Client can send a single fetch request that would select many Topics then there could be a problem with queueing too many fetch replies at once for a Client as the Client queue could fill and the Client get disconnnected.

To avoid this, when a single fetch of a large number of Topics is expected then 'fetch batching' can be used. This allows fetches to be performed in batches with a specified time interval in between thus allowing the Client to process the Messages and mitigate the possibility of the Client queue filling.

The facility is enabled by configuring a 'Fetch Policy' for the Connector that the Clients connect to. The policy may be configured in the *`etc/Connectors.xml`* configuration file or using the `FetchPolicyConfig` configuration object obtained from the connector. The policy comprises two settings, the 'batch size' and the 'delay'.

The batch size specifies the number of single Topic fetch requests to be performed in a batch before waiting for the delay period to execute the next batch. If the batch size is specified as 0 then no batching will occur.

Fetch Handlers

It is possible to assign a 'fetch handler' to a Topic. Such a handler will be be called whenever a Client attempts to fetch the Topic.

The handler is called after any other authorisation processes and may be used to provide an additional layer of authorisation.

Copyright 2013 Push Technology

It is also possible to delegate the fetch to some asynchronous process and defer replying until a response is received (see below).

If a fetch handler is in use for a Topic it will always be used in preference to obtaining the Topic Data state or invoking the Publisher to obtain the state. For this reason it would not normally be used for Topics that do have Topic data.

The following example shows a fetch handler being used to prevent mobile clients from fetching the content a Topic. Note that in this case the handler has been applied to a Topic that has TopicData and so the actual state can be obtained from the TopicData:-

```
        TopicDefinition topicDef =
            new TopicDefinition(
                TopicDataType.SINGLE_VALUE,
                MetadataFactory.newFieldMetadata(MDataType.STRING));

        topicDef.setProperty(
            TopicProperty.FETCH_HANDLER,

            new TopicFetchHandler() {

                @Override
                public TopicMessage fetchForClient(
                TopicClient client,
                Topic topic,
                List<String> headers) {

                    ConnectionType type = client.getConnectionType();
                    if (type.isCategory(ConnectionCategory.MOBILE)) {
                        LOG.warn("Mobile clients not permitted");
                        return null;
                    }
                    try {
                        return topic.getData().getLoadMessage();
                    }
                    catch (Exception ex) {
                        ex.printStackTrace();
                        return null;
                    }
                }

            });

        addTopic("NonMobile",topicDef);
```

Asynchronous Fetch using Fetch Handler

Another use of fetch handlers is to delegate the fetch to some aynschronous process and then returning the reply later.

The following example shows the use of a fetch handler to delegate the fetch to an Event Publisher and then send the reply when the Publisher gets a response.

```
    @Override
    protected void initialLoad() throws APIException {

        TopicDefinition topicDef =
            new TopicDefinition(
                TopicDataType.SINGLE_VALUE,
                MetadataFactory.newFieldMetadata(MDataType.STRING));

        topicDef.setProperty(
            TopicProperty.FETCH_HANDLER,
            new TopicFetchHandler() {
```

Copyright 2013 Push Technology

```
                @Override
                public TopicMessage fetchForClient(
                TopicClient client,
                Topic topic,
                List<String> headers) {
                    try {
                        TopicMessage message =

theEventPublisher.createDeltaMessage("FetchTopic");
                        message.setHeaders(headers);
                        message.putFields(
                            client.getClientID(),
                            topic.getName());
                        theEventPublisher.send(message);
                    }
                    catch (Exception ex) {
                        ex.printStackTrace();
                    }
                    return null;
                }
            });

        addTopic("Delegate",topicDef);
    }

    @Override
    protected void messageFromEventPublisher(
    EventConnection eventConnection,
    TopicMessage message) {
        if (message.getHeader(0).equals("FetchReply")) {
            try {
                Client client = Publishers.getClient(message.getHeader(1));
                if (client!=null) {
                    List<String> correlationId = new ArrayList<String>();
                    correlationId.add(message.getHeader(2));
                    client.sendFetchReply(message,correlationId);
                }
            }
            catch (Exception ex) {
                LOG.error("Error handling fetch reply",ex);
            }
        }
    }
```

The example shows the fetch handler sending a request to an event publisher containing the client id and the topic to fetch. It then returns null so that no reply happens at this point.

Topic Sets

'Topic Sets' are a mechanism in the Java API for specifying a set of Topic names and/or Topic Selectors.

Topic Sets are provided by the issued TopicSet class.

TopicSet extends the Java LinkedHashSet class and thus provides a set which maintains the order that elements were added to it. This can be important where the order that Topic selectors are applied is important.

TopicSet also provides a validation mechanism. Its validate method can check that all entries are valid Topic names and/or selectors.

Most API interfaces to which Topics may be specified also have equivalent methods that will take Topic Sets.

Copyright 2013 Push Technology

Topic Attachments

Attaching an object to a Topic

In the Java API every Topic may have an 'attachment'. An attachment is an object which the user may associate with the Topic.

An attachment may be attached to a Topic using the `attach` method and retrieved using the `attachment` method. To remove an attachment `attach(null)`.

If a Topic attachment is relied upon for the Topic state then it should be attached when the Topic is created using the addTopic variant that takes a `TopicDefinition`.

Topic aliasing may be turned on or off for a Publisher using a *`etc/Publishers.xml`* entry. For an Event Publisher it can be turned on or off using a `setTopicAliasing` method.

Though the data for a Topic may be attached to a Topic in this way the preferred method would be to use Topic Data.

Copyright 2013 Push Technology

Chapter 11

Messages

Topics:

- *Introduction*
- *Message Types*
- *Creating Messages*
- *Populating Messages*
- *User Headers*
- *Reading Messages*
- *Records*
- *Metadata*
- *Byte Encoding*
- *Character Encoding*
- *Message Priority*
- *Acknowledged Messages*
- *Fragmented Messages*
- *Message Filters*

Messages are fundamental to Diffusion™.

Publishers publish Messages on Topics which Clients subscribe to so they can receive those Messages. Clients can also send Messages to Publishers. Messages are sent from Event Publishers and routed to the Publishers that should receive them. Messages are part of the basic Diffusion™ Protocol and have the same underlying format no matter what platform or API is being used.

Introduction

Within user written applications Messages must be created and populated with data for publishing or sending. The recipients of those messages can then read them.

Special features of Messages include the ability to change the delivery priority or request that they be acknowledged by the recipient.

Within Publishers there are special facilities for comparing Messages.

What is a Message?

The make-up of a message

In Diffusion™ terms a Message is a series of bytes with a variable length header. Its format is as follows;

Table 37: Message Format

Message Length	The length of the Message including headers. This is a 4, 2 or 1 byte (space used is configurable) signed twos-complement integer.
Message Type	A single byte identifies the type of Message.
Byte Encoding	A single byte indicates whether byte encoding is applied to the Message.
Fixed Headers	Optional Fixed Headers depending upon Message Type.
User Headers	User Headers which may be added to a Message before adding data.
Data	Optional Data.

Headers

The first 6 (4 or 3) bytes is a fixed overhead for all Messages.

Whether there are any Fixed Headers depends upon the Message type (see Protocol Specification for full details). User Headers may be added as required.

Headers are variable length and are separated by 'Field Delimiters' (byte value x02) and headers as a whole are terminated by a 'Record Delimiter' (byte value x01). These delimiters are special values used within Diffusion™ Messages.

Headers are UTF-8 encoded character strings. Use of UTF-8 means that the characters used in header values (such as Topic Names) are not restricted to the ASCII set.

Data

The data part is a variable number of bytes. Diffusion™ has no interest in the content which is entirely user specific. It can be character data, encoded into bytes in a way that the user requires, or it can be any form of binary data. There are no restrictions on data content except for Browser clients using Webscoket, XHR or Iframe in which messages are UTF-8 encoded.

The APIs provide facilities for writing data to Messages and reading data from Messages.

Message Record and Fields

Diffusion™ has the concept of Records and Fields within a Message

Even though the data within a Message is user specific, Diffusion™ has the concept of Records and Fields within a Message which may simplify Message handling, particularly within Publishers.

Message Capacity

Specifying the capacity of a message

Copyright 2013 Push Technology

Within the Diffusion™ Server (and some APIs such as Java), Messages have internal Message buffers. A Message buffer is a fixed length area of memory assigned to make the handling of Messages more efficient.

Whenever a Message is created in Java, a Message capacity should be specified (a default is taken if it is not). The capacity specified would only need to cover the data and user headers (other overheads would be automatically calculated). To avoid unnecessary memory usage the capacity should be as small as possible. If more data is written to a message than was specified to its capacity then the message will be automatically extended but as there is an overhead to such extensions the capacity should be sufficient for maximum performance.

It is important to note that the Message size and the Message capacity are different and only the number of bytes indicated by the size are ever sent over connections even though each Message will be consuming its capacity (plus an allowance for headers) in memory whilst it exists within the Server (or Java API) environment.

In-bound Messages are assigned buffers which exactly match the message size in their headers.

When a Message is encoded then a second internal buffer is also assigned to the Message. An in-bound encoded Message would only have an encoded Buffer until it is actually read at which point a decoded Buffer is assigned.

Maximum Message Size

The concept of maximum message size exists within the server environment and within client side APIs

This is a system wide setting which specifies the largest size that any message can be (including headers). This value is also used in certain areas for tuning.

Within the server the maximum message size is specified in *etc/Server.xml*

In client side Java code the maximum message size is as specified using `ConfigManager.getConfig().setMaximumMessageSize()` which should be called before a connection is made.

Message Types

The different types of message specified by the Diffusion protocol

The Diffusion Protocol specifies a number of different Message types. Unless a user is writing an application using the raw protocol then most of these Message types will be of no concern as they are largely hidden by the APIs. However, it may be of use to be able to identify the Message types for diagnostic purposes.

This section discusses only those Message types that are made visible via the APIs

Topic Messages

The types of messages that are published on Topics

Messages that are published on Topics are one of two types - 'Topic Load' and 'Delta'. Both of these types comprise a Fixed Header containing the Topic name, optional User Headers and data (which though typical is also optional). These two types are discussed below.

Topic Load Messages

A 'Topic Load' message is what is (usually) sent to a Client when it subscribes (or resubscribes) to a Topic. The purpose of it is to provide the Client with the current state of the Topic data. The use of these is not mandated by Diffusion but provides a simple mechanism for dealing with what is typical functionality.

Because these messages represent the Topic state they are typically only created by a Publisher but could also be sent to a Publisher from an Event Publisher if required.

Such messages would typically be created during Topic Loading. See the section on creating Messages for more detail.

In the Java API this would be a `TopicMessage` which returns `true` to the `isTopicLoad` method.

Delta Messages

Copyright 2013 Push Technology

The word 'delta' is conventionally used to describe 'change'. A Delta Message is therefore typically used by a Publisher to publish changes in the state of a Topic to all Clients that are subscribed to that Topic.

Delta messages may also be used by Publishers to send Messages to individual Clients, and by Clients to send Messages to Publishers.

A Delta Message does not need to contain the whole Topic state when a change occurs. Ideally, only those items of data that have changed need to be sent. The user needs to define some internal application protocol to handle this when designing a Publisher. The use of hierarchical Topics can simplify this if only a single data item is stored with each Topic in the hierarchy.

How Delta Messages are created depends upon which API is in use. See the section on creating Messages for more detail.

In the Java API this would be a `TopicMessage` which returns true to the `isDelta` method.

Ping Messages

Ping Messages are used to test a connection exists, and it's latency

Ping Messages are special types of Message used to 'ping' the other end of a connection in order to establish that it is still there and also to measure latency. When one end of a connection sends a Ping Message it is the responsibility of the other end to reflect it back as soon as it is received.

It should be noted that this Ping Message handling is all handled by the APIs which provide mechanisms for sending such a Message and being notified when its response is received

Server Ping

A Server Ping Message is one sent from a Client to a Server which attaches the Client queue size to it before reflecting it back to the Client.

Client APIs will provide a mechanism for sending such a Ping Message and receiving the reply.

For example in the Java Client API a Server Ping can be sent using the ExternalClientConnection.ping method and the response will be returned to any ServerPingResponseListener set using the setPingResponseListener method.

Client Ping

A Client Ping Message is one sent from a Publisher to a Client which then reflects it back so that the Publisher can measure the round trip time. The Client API will automatically reflect such messages and therefore the Client application does not need to handle them. See 'Client Pings' in the Publisher section for details of how to initiate a Client Ping and receive its response

Acknowledged Messages

a Publisher can set a Message as 'requiring acknowledgement'

Acknowledged Messages are actually different Message types in protocol terms because they carry an extra header holding the ACK Id. However from the point of view of the APIs this will not be apparent.

For example, in the Java API, when a Publisher sets a Message as 'requiring acknowledgement' its underlying Message type is changed and an extra header is added, however, from the point of view of the API it is the same Message. Upon receiving such Messages the Client APIs strip off the ACK id before converting the Message type and passing the message onto the Client application. The same occurs in the opposite direction (from Client to Server).

Control Messages

Messages which are unseen by users of the API

There are a number of other control messages within the protocol which are never seen by users of the API. For example the Messages sent from a Client to the Server when it subscribes to Topics.

For diagnostic purposes the various types of control Message are all described in the Protocol Specification.

Copyright 2013 Push Technology

Creating Messages

In protocol terms a Message is a stream of bytes sent using the Diffusion™ Protocol

In API terms a Message is an object which can be populated and sent or received and read. Different APIs will approach Message object creation in different ways. This section will use the Java API as an example. For other APIs consult the specific API documentation.

Publisher Factory Methods

The mechanism for creating Topic Messages are factory methods on various API classes. These allow you to create either Topic Load (using createLoadMessage) or Delta (using createDeltaMessage) Messages.

In a Publisher environment such methods exist on the Publisher, the Topic or there are static versions on the Publishers class.

The Topic that the Message is for must always be specified when creating a Topic Message.

When a Message is created an internal buffer is assigned to it and therefore the initial size of the message data (its capacity) should be specified on creation.

A Topic may have default load and delta Message capacities set so that it is not necessary to specify capacity on every Message creation. If the capacity is not specified then configured defaults are used.

Where Message capacities are the same or similar across a number of Topics it may be more flexible to specify the capacity as a value obtained from a Publisher integer property.

Topic Load and Delta messages can also be created with properties allowing them to be fragmented when sent from a Publisher. Using createLoadMessage or createDeltaMessage, a fragment size may be specified. If the message exceeds this size, it may be split into smaller fragments which are sent individually and recombined by the client. Additionally, a delay between sending each fragment should be given to allow large messages made of many fragments to be interleaved with other messages. A delay of zero indicates that the messages should be sent as quickly as possible.

Using API Factory Methods

In APIs other than the Publisher API there are various factory methods for creating messages as follows;

Client APIs

Both the ExternalClientConnection (Client API) and PublisherServerConnection (Publisher API) represent Client connections to a Diffusion™ Server and as such are both of type ServerConnection.

A ServerConnection has a convenience method called createDeltaMessage to create Messages. It is normally best to supply a capacity when creating Messages to avoid unnecessary automatic extension of message.

Event Publisher API

The EventPublisherConnection class has createLoadMessage and createDeltaMessage methods which operate in a similar way to the Client API.

Populating Messages

How to populate messages with user header and/or user data.

A Diffusion Message comprises Diffusion headers, user headers and user data. (see Message Basics). So when using an API, after creating a Message, it then needs to be populated with user header and/or user data.

User headers are optional and there may be any number of them as required. They must be added to the message before any data is added. Headers are UTF-8 encoded character data.

Copyright 2013 Push Technology

Data is 'byte' data and therefore can be used for any purpose that the user wishes. If the user wishes to write character data then it would typically be UTF-8 encoded but any character encoding could be used if desired. Diffusion has no interest in the data content but the APIs do provide some utilities that make populating and reading Message data much easier.

Diffusion provides the facility for compressing or encrypting Message data for transmission. This occurs at a byte level (i.e. the bytes within the Message have compression or encryption applied to them) - this is referred to as the Message's 'byte encoding'.

Each language API may address the populating and encoding of messages differently. The remainder of this section discusses populating Message using the Java API (i.e. within a Publisher, or when using other external Java APIs).

Setting User Headers

User Headers may be set on a Message using the setHeaders method which is a varargs method and thus any number of String headers may be set in one call. This may only be called once and must be called before any data is added to the Message. There is also a version of this method that takes a List parameter.

```
TopicMessage message = Publishers.createDeltaMessage("MyTopic",20);
message.setHeaders("Header1","Header2");
```

Setting Data

In the Java API data is put into Messages using the various 'put' methods. When a Message is newly created it has a pointer set just after the Message headers and a put will write data at the pointer position and then update the pointer position.

It is important to note that once a Message has been published (or sent) it can no longer be updated.

Setting String Content

If you want to write character data to a message then the put(String...) method can be used. As this is a varargs method you can supply any number of Strings and they will be concatenated in the message. so, for example

```
TopicMessage message = Publishers.createDeltaMessage("MyTopic",6);
message.put("AA","BB");
message.put("CC");
```

Each call to 'put' updates a pointer in the Message so that the next 'put' (or any other Message populating method) will start from where the last one ended. So the above example results in a Message with data content of:-

AABBCC

It should be noted that what is actually written to the Message is the byte representation of the Strings encoded using the default character set for the Message (see character encoding). So, the number of bytes used in the Message could be more than the number of characters put.

Setting Byte Content

For non character data you can write bytes directly to the Message using either the put(byte) or put(byte[]) methods.

Using Records and Fields

To simplify the handling of character data, Diffusion™ has the concept of 'Records' and 'Fields'. See the section on 'Records' for more information.

Copyright 2013 Push Technology

User Headers

Allows String header values in a message.

User Headers are a feature of Diffusion™ Messages that allows String header values to be set in addition to and separately from the Message's data payload. They offer a convenient mechanism for adding extra control information to a message.

User Headers are typically set after creating a Message and before populating with other data. It is permissible for a Message to have only User Headers and no data.

In terms of the Message content User Headers are held after(and in the same format as) any Fixed Headers.

User Headers are String format and are UTF-8 encoded thus supporting the full Unicode character set.

User Header values may be set in a message using the APIs. Each API may differ in the way that this is done.

The use of User Headers within the Java API is outlined below.

Adding User Header Data

User Header values must be set after creating a Message and before populating it with any data as follows

```
TopicMessage message = Publishers.createDeltaMessage("MyTopic",10);
message.setHeaders("H1","H2");
message.put("data");
```

Retrieving User Headers

User Header values must be set after creating a Message and before populating it with any data as follows

```
TopicMessage message = Publishers.createDeltaMessage("MyTopic",10);
 message.setHeaders("H1","H2");
 message.put("data");
```

Retrieving User Header Data by Index Number

It is also possible to access header values individually

```
String h2 = message.getHeader(1);
System.out.println(h2);
```

Note that header index numbers start from 0 so the second User Header value is obtained using index number 1. If a Header with the requested index does not exist then null is returned.

Reading Messages

The mechanisms for reading the content of Messages

Diffusion Messages are typically created and populated by Publishers and then published to Clients that then read the Message content. Messages may also be created by Clients (or other APIs such as the Event Publisher) and sent to the Publisher.

Copyright 2013 Push Technology

Whether within a Client or a Publisher the various APIs provide mechanisms for easily reading the content of Messages. The way in which this is done will vary across different APIs.

This section outlines how message content can be read in the Java API, for example within Publishers or when using the Java Client API.

Reading Messages in the Java API

The data content of a Message (as opposed to the User Headers) can be read in a number of different ways. The most suitable to use will depend upon the nature of the data and how it has been structured within the Message.

Encoding Considerations

Note that though Messages may have been byte encoded for transport, before they are read they are automatically decoded therefore from the point of view of the API user there is no need to consider the fact that the underlying message could be encoded.

When reading character data (Strings) from Messages the character encoding may need to be considered. Character reads will automatically convert Message bytes to characters using the Message's defined character set. As the default is normally UTF-8 then no special handling of character sets is usually required.

Message Pointers

Message data reads can be 'absolute' or 'relative'. With an 'absolute' read the data is read positionally from the Message. With a 'relative' read, data is read relative to a 'pointer' maintained within the Message. When an incoming Message is presented to an API by Diffusion™ then its pointer will be set at the start of the Message data. Each relative read will move the pointer to be positioned after the data read. Absolute reads have no effect on the pointer. The pointer may be reset to the start of the data at any point using the rewind method.

Reading Whole Message Content

The whole data content of a Message may be read as a String or as an array of bytes.

Content as a String

To read as a String

```
String data = message.asString();
```

The above call will convert all of Message data bytes to a String using the Message's default character set.

It is important to note that the length of the returned String may not be the same as the length returned by the length method of the Message. This is because only standard (e.g. ASCII) characters can be encoded as a single byte, so if non standard characters are in use the returned String length could be shorter than the Message length.

Content as bytes

To read a Message as bytes

```
byte[] bytes = message.asBytes();
```

In this case the byte array returned will be exactly the same length as the value returned by the Message length method.

Both of the above examples are absolute reads and do not affect the Message pointer.

Copyright 2013 Push Technology

Traversing Message Content

It is possible to traverse the content of a Message in a relative manner using the 'next' set of methods on the Message. Each time one of these methods is used the Message pointer is updated so that it points to after the data that has been read.

Reading Bytes

To read a Message byte by byte the nextByte method may be used. To read blocks of bytes the nextBytes method can be used. The latter may be useful for fixed format Messages. In both cases an exact number of bytes will be read starting from the current Message pointer.

The remaining method may be used to determine how many bytes of data there are left to read in the Message at any time. The hasRemaining method indicates whether any bytes remain to be read.

It is important to determine whether there are sufficient bytes to read before invoking nextByte(s) as an exception will occur if an attempt is made to read off the end of the Message.

Reading One Byte

The following example shows how to read Message data one byte at a time. In this example each byte value is displayed as a decimal value.

```
byte b;
while (message.hasRemaining()) {
 b=message.nextByte();
 System.out.println(Byte.toString(b));
}
```

Reading Array of Bytes

The following example shows how to read a message in 4 byte segments and display the Hexadecimal values of each 4 bytes using the Diffusion™ Utils class

```
byte[] bytes = new byte[4];
while (message.remaining()>3) {
 message.nextBytes(bytes);
 System.out.println(Utils.bytesToHex(bytes,0,4));
}
```

In both of the above examples it is assumed that the Message pointer is positioned at the start of the data as it would be had the Message been freshly presented to an API by Diffusion™ or if message.rewind() had been called.

Reading Records and Fields

If a Message has been populated using the Diffusion™ Field and/or Record delimiters then the nextRecord and nextField methods make the reading of these Records and Fields simple. See the section on 'Records' for more information.

Concurrency Issues

Copyright 2013 Push Technology

When working in a multi-threaded application it is important to note that the relative methods of a Message are NOT threadsafe. If more than one thread accesses a Message at the same time then they may both be updating pointers at the same time. One way of overcoming this may be to take a copy of the Message (using TopicMessage.duplicate) but this is inefficient and uses additional memory. A better mechanism is to use 'Message Readers' (see below).

Using a Message Reader

To read a message safely within a thread when it might also be accessed in another thread a MessageReader may be used. A MessageReader is an object that maintains its own read-only view of a Message without affecting that Message's pointers.

To obtain a MessageReader use the getReader method on a Message. The Reader has all of normal Message read methods so you can access the Reader in the same way as if it were the real Message.

Note that a Reader will maintain a reference to the Message it was obtained from thus preventing the garbage collection of that Message whilst the reader is still in use.

Records

Even though the data within a Message is user specific, Diffusion™ has the concept of Records and Fields within a Message which may simplify Message handling, particularly within Publishers.

Records and Fields relate to the handling of character based Messages as they depend upon the use of separator bytes which are outside the character representation range. They could not be used with binary data..

The separators used are

Table 38: Separator Bytes

Field Delimiter	Single byte value of Hexadecimal 02
Record Delimiter	Single byte value of Hexadecimal 01

A notation of <FD> for Field Delimiter and <RD> for Record Delimiter may be seen throughout this manual

A 'Field' is defined as zero or more characters terminated by <FD> , <RD> or the end of the Message

A 'Record' is defined as zero or more characters terminated by <RD> or the end of the Message

In either case, if a Message ends with an <FD> or an <RD> it implies an empty Field or Record at the end of the Message

This representation is chosen to simplify the splitting up of character data.

Note that because of the format it is not possible to distinguish between an empty Message and one with a single empty Field or a single empty Record and this must be borne in mind when designing Message formats.

The Java API provides Record and Field handling capabilities that greatly simplify the handling of character based Messages. Also, when using the Java API, Records may have Metadata associated with them allowing fields to be populated and read by name.

The delimiters are handled as separators and not terminators so that they can easily be used to split up a String that contains them (see 'Reading Messages'). Because of this it should also be noted that there is no difference between the internal representation of an empty Message and a Message with a single empty Record and therefore the use of such delimiters should never assume that this distinction could be made.

It should also be noted that Records are handled as separate objects and are not a map onto the Message itself. This means that they can be manipulated between 'puts' which can simplify the handling of data. Changes to a record after putting it into a Message are not reflected in the Message.

Unlike Messages, Records have methods allowing Fields to be inserted, removed and replaced.

Copyright 2013 Push Technology

Populating Messages

The Java API handles Records and Fields as follows.

Adding fields to a message

Fields provide a further level of Structure to a Message

'Fields' are Strings which are written to the Message separated by field delimiter bytes. Using these field delimiters separates the character data into variable length String fields within the Message.

So, contrast the example of setting String content with the following example of using fields

Adding fields to a TopicMessage

```
TopicMessage message = Publishers.createDeltaMessage("MyTopic",10);
message.putFields("AA","BB");
message.putFields("CC");
```

The resultant TopicMessage will consist of 3 fields. The field delimiter that is used is defined by the constant Message.FIELD_DELIMITER. If we represent the field delimiter as `<FD>` (though it occupies only one byte) then what is actually written to the message in this case is:- `AA<FD>BB<FD>CC`

In this case a total of 8 bytes are written to the Message. Note that there is no delimiter at the start of the Message data nor at the end. Writing fields in this manner greatly simplifies reading the data from the Message and avoids the space wastage of using Message definitions with fixed length fields.

Adding records to a message

Records provide a further level of Structure to a Message.

Records are best thought of as groups of Fields separated by a special record delimiter.

This provides for the ability to easily handle structured data as records.

For example, a Message could contain a variable number of 'customer' records, each containing 'Customer Number' and 'Customer Name'. You could write such a Message as follows

Adding records to a TopicMessage

```
TopicMessage message = Publishers.createDeltaMessage("MyTopic",40);
 message.putRecords(
  new Record("123","John Smith"),
  new Record("124","Fred Bloggs"));
```

The record delimiter that is used is defined by the constant Message.RECORD_DELIMITER. If we represent the record delimiter as `<RD>` (though it occupies only one byte) then what is actually written to the message in this case is:- `123<FD>John Smith<RD>124<FD>Fred Bloggs`

As with Fields, note that there is no delimiter at the start or end of the Message

Adding a record without using a Record object

Records can be manipulated as a separate object or can be added easily to a Message using the putRecord(String...) method.

Add multiple fields as a record

```
message.putRecord("125","A.N. Other");
```

Copyright 2013 Push Technology

Building upon the previous example we now have a Message containing `123<FD>John Smith<RD>124<FD>Fred Bloggs<RD>125<FD>A.N. Other`

Adding an empty record to a message

It is permissible to put empty records into a Message.

Add multiple fields as a record

```
message.putRecords(new Record(),new Record());
```

We now have a Message containing `23<FD>John Smith<RD>124<FD>Fred Bloggs<RD>125<FD>A.N. Other<RD><RD>`

Note that in this case the final `<RD>` implies a blank record after it.

Reading Messages

If a Message has been populated using the Diffusion™ Field and/or Record delimiters then the nextRecord and nextField methods make the reading of these Records and Fields simple

Getting fields from a message

If the Message is made up only of Fields then the Fields may be processed in order using nextField

```
String field;
while (message.hasRemaining()) {
 field=message.nextField();
 System.out.println("Field="+field);
}
```

Note that the above examples is not a reliable way of processing if there is the possibility of empty Fields at the end of Messages

Getting records from a message

When Records are in use then the Records can be read one at a time using nextRecord

```
Record record;
while (message.hasRemaining()) {
 record=message.nextRecord();
 System.out.println("Record="+record);
}
```

Note that the above examples is not a reliable way of processing if there is the possibility of empty Records at the end of Messages. One should look at *Catering for Empty Fields and Records* to solve this problem

Looping through Records and Fields

Records also provide a convenient way of accessing data positionally as the Fields of each record read can be accessed by their index. The following example demonstrates this by displaying only the content of the third Field of each Record read

```
while (message.hasRemaining()) {
 record=message.nextRecord();
```

Copyright 2013 Push Technology

```
 if (record.size()>2) {
  System.out.println("Field 3="+record.getField(2));
 }
}
```

We now have a Message containing `23<FD>John Smith<RD>124<FD>Fred Bloggs<RD>125<FD>A.N. Other<RD><RD>`

Note that in this case the final `<RD>` implies a blank record after it.

Catering for Empty Fields and Records

Handling empty fields or records

The method `hasRemaining` determines whether the end of the Message has been reached. Because this method indicates whether there are any bytes left to read in the Message it will work as long as there is not the possibility of there being an empty Field or Record at the end of the Message.

An empty Field is one that is of zero length and thus the last byte of a Message could be a Field delimiter implying the presence of an empty Field after it. However, as the nextField method will always position after the last field read then hasRemaining would not detect the presence of an empty Field at the end of a Message.

An empty Record is one that has no fields within it. A record delimiter at the end of a Message therefore implies an empty Record following. As with Fields, checking for end of processing with hasRemaining would not detect an empty Record at the end of the Message.

Both nextField and nextRecord return null when there is no more to process. If there is an empty Field at the end of a Message then nextField would return a zero length String (even though hasRemaining would return false). If there is an empty Record at the end of a Message then nextRecord will return an empty Record (i.e. one with zero Fields).

Traversing fields

Traversing the Fields of a Message when the last Field could be empty.

```
while ((field=message.nextField())!=null) {
 System.out.println("Field="+field);
}
```

Getting records from a message

Traversing the Records of a Message when the last Record could be empty

```
while ((record=message.nextRecord())!=null) {
 System.out.println("Record="+record);
}
```

Metadata

Metadata is 'data that describes data' and in Diffusion™ terms it is something that defines the format of a Diffusion™ Topic Message.

Diffusion™ Metadata is a generic mechanism for describing Message formats regardless of the exact data representation. It describes a Message in terms of 'Fields' within it and these fields may be grouped into 'Records' which themselves may be further subdivided into Fields and/or Records.

Copyright 2013 Push Technology

This generic representation of Metadata can potentially be used for many different types of Message data. For example, Metadata may be used to describe a Diffusion™ Record based Message where Fields and Records have the same meaning but certain constraints exist (e.g. nesting of Records is not permitted).

The purpose of Metadata is to allow for programatical modelling of data structures. Currently it is only available within the Java API and used in the following ways

- To define the layout of a Record so that fields within the Record can be addressed by name
- To define the layout of the Messages used by Record Topic Data in terms of one or more Record definitions.

Metadata Structure

Definition of Messages, Records and Fields for Metadata

A Metadata definition is made up of a hierarchy of Metadata 'nodes' (class MNode) with a Message node (class MMessage) at the top. A Message is defined as one or more Field (class MField) and/or Record (class MRecord) nodes. A Record is also defined as one or more Field and/or Record nodes (thus MMessage is a specialisation of MRecord). A Field defines an elementary data item of a particular data type

Every node has a name which must be unique within its parent node. Every child node may represent one or more possible occurrences of that Record or Field within the parent. The number of possible occurrences of a node is described by its multiplicity.

The order in which Nodes are defined within their parent defines the order that they will appear in within a Message.

Message Metadata

A Metadata Message (class MMessage) is the top level object of a Metadata definition which represents a whole Message layout.

Metadata Messages are created using the MetadataFactory class specifying the type of Topic Data that the Metadata describes. Even though Metadata definitions are generic in form, the type of data may impose constraints on the Metadata.

A Message may be made up of one or more Fields and/or Records

Metadata Record

A Metadata Record (class MRecord) defines a group of Fields and/or Records within a parent Message or Record.

A Record has a multiplicity within its parent node.

Field Metadata

A Metadata Field (class MField) defines an elementary data item within a Message or Record.

Every Field has a data type which defines its actual representation within the Message.

A Field may have a default value specified to use when initialising Message data.

A Field has a multiplicity within its parent node.

Data Types

The data type of a Field defines its actual representation within the Message. The data types available will vary according to the Topic Data Type that the Metadata is defining. The same data type may result in different physical data representations for different Topic Data Types.

The available data types are defined by the enum MDataType and the following are currently available

Table 39: Data Types

STRING	A character string. Represented internally as a java.lang.String.

Copyright 2013 Push Technology

	The initial default value for this type is "" (a zero length String).
INTEGER_STRING	An integral number represented in the Message as a character string. This is represented internally as a java.math.BigInteger. If a Field is defined as this type then it can only contain numeric digits with an optional leading sign. By default, empty fields are not allowed. The initial default value for this type is "0".
DECIMAL_STRING	A decimal number represented in the Message as a character string. This is represented internally as a java.math.BigDecimal. Decimal fields have the number of places to the right of the decimal point defined by the 'scale', the default being 2. Such values may be parsed from a character String with any number of digits to the right of the decimal point but 'half up' rounding will be applied to achieve the target scale and output of the field will be rendered with the specified scale. By default, empty fields are not allowed. The initial default value for this type is "0.00" (depending on scale). For comparison purposes the scale is ignored therefore a value of 1.50 is the same as 1.5.
CUSTOM_STRING	This is a special type where the behaviour is delegated to a user written CustomFieldHandler This type is available in all Topic Data Types

Scales

Decimal format Fields (such as DECIMAL_STRING) may have a scale which defines the number of places to the right of the decimal point.

The scale of a Field is set using the setScale method. If not specified then 2 is assumed. This value is ignored for all but decimal format fields.

Default Values

Every field may have a default value specified for it using the setDefaultValue method. If not specified then the default value is assumed to be the default for the data type.

When a Message is created using Metadata then default initialisation will apply the default values specified for each Field.

Empty Values

By default STRING type fields will accept an empty field (i.e. a zero length string) as input but other types do not. However it is possible to allow other field types to allow empty input also. This is done using the setAllowsEmpty method on the MField object.

Multiplicity

The number of times the corresponding data can occur within its parent

Copyright 2013 Push Technology

The 'multiplicity' of a Metadata Field or Record defines the number of times the corresponding data can occur within its parent

Multiplicity (defined by the Multiplicity class) is defined in terms of the minimum and maximum number of occurrences. Some data representations (such a Protocol Buffers) support variable numbers of nodes, whereas others (such a Record Data) only support fixed number of nodes (where minimum=maximum) except in the last position.

Fixed multiplicity can therefore be defined by a single number. For example a multplicity of 5 (e.g. new Multiplicity(5)) indicates that there must be exactly 5 occurrences of the node within its parent.

Variable multiplicity is defined in terms of a minimum value and a maximum value and is represented with the notation n..n. So a multiplicity of 1..5 (new Multiplicity(1,5)) indicates that there can be between 1 and 5 occurrences of the node within its parent. A special maximum value of -1 is used to represent no maximum so a multiplicity of 1..n (new Multiplicity(1,-1)) means that there can be any number of occurrences of the node but there must be at least one.

Optional nodes are indicated by a a minimum value of 0. So 0..1 would represent a node that can occur 0 or once and 0..n would represent a node that can occur any number of times or not at all. Of course a fixed multiplicity of 0 is not allowed

Examples

Creating a Custom Field Handler

A Custom Field Handler can be used for defining a data type other than those provided and can deal with any String data format required

The following example shows a handler used to represent a Double value

```
public class DoubleFieldHandler implements CustomFieldHandler {

 public Object getInitialDefaultValue() {
  return new Double(0.0);
 }
 public Object parse(Object object) throws APIException {
  if (object==null) {
   return new Double(0.0);
  }

  try {
   return (Double.parseDouble(object.toString()));
  }
  catch (Throwable ex) {
   throw new APIException(
   "Unable to parse "+object+" as double value",
   ex);
  }
 }

 public boolean areEqual(Object source,Object target) {
  return source.equals(target);
 }
}
```

Defining a Record Metadata

The generic Metadata facility may be used to programatically define a Metadata message definition which can be used when creating Records

The following example shows Message Metadata being defined with a single Record within it:

Copyright 2013 Push Technology

```
MMessage messageMetadata =
 MetadataFactory.newMetadata("MyMessage",TopicDataType.RECORD);

MRecord recordMetadata = messageMetadata.addRecord("MyRecord");
recordMetadata.addField("Name");
recordMetadata.addField("AccountNumber",MDataType.INTEGER_STRING);
recordMetadata.addField(
 "Price",
 MDataType.DECIMAL_STRING,
 new Multiplicity(4));
```

The above example shows a Message being defined as a single Record defined as a String Field called "Name", followed by an integer Field called "AccountNumber", followed by decimal Field called "Price" which has a multiplicity of 4 (i.e repeats exactly 4 times).

There are some rules on the Metadata used for Records as follows

- A Message can have one or more Records within it
- Each Record can have one or more Fields defined for it
- Only String data types are permitted for Fields (STRING, DECIMAL_STRING etc). 'Custom' data types may also be defined which can implement special behaviour for fields.
- Repeating Field definitions are permitted but only with fixed multiplicity (e.g. you can have a multiplicity of 4 but not 1..4). The exception to this is the last field within the record which can have variable multiplicity (e.g. 0..n). When Records are used within a Message then only the last Record within the Message can have a repeating Field at the end and the Record itself may not repeat.
- Fields may not exist directly under a Metadata Message, only within a Record. Even though the Message interface does permit Messages containing only Fields, a single Record is implied in those cases
- Repeating Record definitions are permitted within a Message but only with fixed multiplicity. The exception to this is the last Record within a Message which can have variable multiplicity.
- Nested Records are not permitted (i.e. you cannot define a Record within a Record).

A single Metadata definition may be reused for any number of Records. To allow reuse of the same definition at both the Client and the Publisher it is recommended to encapsulate Metadata definitions in their own Java classes.

Creating and Populating a Record with Metadata

Once you have a Metadata definition it can be reused to create and populate Records

This example assumes the Metadata has already been defined

```
Record record = new Record(recordMetadata);
record.setField("Name","John Smith");
record.setField("AccountNumber",123456);
record.setField("Price","100","98.8","95.25","90");
```

Having set up a Record in this way it may then be written to a Message using the putRecords method. The same Record could then be changed and reused many times if required.

The Fields do not have to be set in any particular order and any that are not explicitly set would take the default values as defined by the Metadata. However, when a Record is written to a Message the Fields will be written in the order specified by the Metadata.

To write more than one occurrence of the same Record type to a Message you put a Record of that type into the Message more than once.

Reading a Record with Metadata

You can read a Record from a Message using Metadata

Copyright 2013 Push Technology

You can then access the fields by name within a Record if the Metadata is specified in the nextRecord method.

```
Record recordIn = message.nextRecord(recordMetadata);

String name = recordIn.getField("Name");
String account = recordIn.getField("AccountNumber");
List<String> prices = recordIn.getFieldValues("Price");
```

If the Record within the Message does not match the Metadata then an exception would occur

Example of using Record Metadata

The example below shows a method that defines some Message Metadata to be used with Record Data. The Message comprises two Records

```
MMessage defineMyMetadata() throws APIException {
 MMessage message =
 MetadataFactory.newMetadata("MyMessage",TopicDataType.RECORD);

 MRecord record=message.addRecord("Record1");
 MField field;
 field=record.addField("Name");
 field=record.addField("AccountNumber",MDataType.INTEGER_STRING);
 field.setDefaultValue(-1);
 field=record.addField(
  "Price",
  MDataType.DECIMAL_STRING,
  new Multiplicity(2));
 field.setScale(3);

 record=message.addRecord("Record2");
 record.addField("AddressLine",new Multiplicity(5));

 return message;
}
```

Loading a Metadata record

Currently Metadata can only be described programatically so it may be useful to employ the singleton pattern to encapsulate Metadata definitions so that they only need to be loaded once.

The following class is an example of such a singleton

```
public class MyMetadata {

 private static MyMetadata theInstance = null;

 private HashMap<String,MMessage> theMetadata =
  new HashMap<String,MMessage>();

 private MyMetadata() throws APIException {
  loadMetadata();
 }

 private static MyMetadata instance() throws APIException {
  if (theInstance==null) {
   synchronized(MyMetadata.class) {
    if (theInstance==null) {
```

Copyright 2013 Push Technology

```
      theInstance=new MyMetadata();
     }
    }
   }
  return theInstance;
 }

 public static MMessage get(String name) throws APIException {
  return instance().getMetadata(name);
 }

 private MMessage getMetadata(String name) {
  return theMetadata.get(name);
 }

 private void loadMetadata() throws APIException {
  loadMessage1();
  loadMessage2();
  //... etc
 }

 private void loadMessage1() throws APIException {
  MMessage message =
   MetadataFactory.newMetadata("Message1",TopicDataType.RECORD);

  MRecord record=message.addRecord("Record1");
  record.addField("Name");
  record.addField("AccountNumber",MDataType.INTEGER_STRING);

  record=message.addRecord("Record2");
  record.addField("AddressLine",new Multiplicity(5));

  theMetadata.put("Message1",message);
 }

 private void loadMessage2() throws APIException {
 // load another message format
 }
}
```

Retrieve a Metadata defintiion

Metadata definitions could then be obtained using the MyMetadata.get(name) method

for example

```
MMessage metadata = MyMetadata.get("Message1");
```

Byte Encoding

Byte Encoding of the Message

Diffusion™ Messages are essentially some header information followed by some byte data and by default this is how the messages are transmitted over a connection.

However, for security and bandwidth considerations Diffusion™ provides a number of 'byte encodings' that can be applied to the data part of a Message when it is sent over a connection. These are as follows

Copyright 2013 Push Technology

Table 40: Types of Byte Encoding

Compressed	The data is compressed using the popular ZLIBcompressionlibrary.
Encrypted	The data is encrypted using an algorithm based on XXTEA.
Base64	The data is encrypted in Base 64 Notation.

The APIs allow the user application to specify the encoding to be used on a per message basis

The actual data encoding and decoding is handled by Diffusion™

Where a Client connects to Diffusion™ using the raw protocol rather than one of the APIs then it should not declare the capability to handle encodings (unless it is able to), in which case the Diffusion™ Server would send out unencoded Messages to such Clients

The use of User Headers within the Java API is outlined below.

Transports capabilities

Table 41: Capabilities of Transports

Transport	Compressed	Encrypted	Base64
DPT	YES	YES	YES
Websocket	NO	NO	NO
HTTP Duplex	YES	YES	YES
HTTP Comet	YES	YES	YES
HTTP	NO	NO	NO
Iframe	NO	NO	NO

Set the Byte Encoding

The default encoding may be specified for a Message after it is created using the setEncoding(Encoding)

```
TopicMessage message = Publishers.createDeltaMessage("MyTopic",10);
setEncoding(Encoding);
```

Character Encoding

Character Encoding of the Message

Diffusion™ Messages are made up of some header information followed by the data payload. From the point of view of Diffusion™ those data are only raw bytes but the representation of characters (normal textual information) as bytes would depend upon the 'character encoding' used. Clearly, whatever character encoding is used when writing data to a Message would also need to be used at the other end of the connection to decode the Message data.

For simplicity it is recommended that the UTF-8 character encoding is used and this is the default for all Diffusion™ API character handling. Message headers are always UTF-8 encoded. The encoding of character data is dependent upon the API but the default should always be UTF-8.

If there is a need to use any other character sets then the default used in the Diffusion™ Server may be changed within the *etc/Server.xml* file.

Copyright 2013 Push Technology

The way in which the character encoding used with Messages is handled may vary across APIs. The handling within the Java API is outlined below.

Character Encoding in the Java API

Character Encoding of the Message

When in a Diffusion™ Server environment (i.e. within Publishers) then the default character encoding that will be used when any String data is written to or read from a Message will be as specified in the *etc/Server.xml* file.

In a Java client side environment the default character encoding can be set using `ConfigManager.getConfig().setCharset()`.

These default settings can be overridden at an individual Message level using the `setCharset` method on the Message.

Using `getCharset` will return the character encoding that the Message will use to encode or decode characters. It should be stressed that it does NOT return the character set that may have been used to encode a Message that has been received over a connection as this information is not passed in the Message protocol.

UTF-8 is the default character set used within Diffusion™.

The Effect of Character Encoding on Message Sizing

Impact of Character Encoding of the Message

An important point to note is that character encoding can result in a discrepancy between the number of data characters written to a Message and the actual number of bytes used to represent that data. This is because certain characters cannot be represented in a single byte. For example, UTF-8 can encode any Unicode character but only some of those (e.g. ASCII characters) can be held as a single byte. Other characters may consume between 2 and 4 bytes.

This should be borne in mind when sizing for Messages that may hold any character data outside the normal ASCII range.

Message Priority

Changing the priority of Messages

Under normal circumstances all Messages published or sent to Clients by Publishers are put on the Client's queue and the Messages are delivered to the Client in the order that they are queued. However, there may be times when a Message needs to be sent to a Client urgently and should go to the the front of the queue or there may be Messages that only need to be delivered when nothing else is queueing. This is acheived by sending or publishing the message at a different priority.

Normal Priority

If a priority is not explicitly specified then 'normal' priority is assumed. This causes a message to be queued after any existing normal priority messages.

All high priority messages will be delivered before normal priority messages are considered for delivery.

All normal priority messages in a client queue will be delivered before low priority messages are considered for delivery.

Only normal priority messages are considered for conflation.

High Priority

A message can be sent with high priority causing it to go to the front of the Client queue.

A Message sent with high priority will be queued after any high priority messages already queued and will be delivered before any normal or low priority messages.

Copyright 2013 Push Technology

A typical use of high priority Messages might be when the Client queue has built up because the Client is not processing Messages fast enough. This might have been notified to a Publisher by means of the 'clientQueueThresholdReached' Client notification.

High Priority Example

The Publisher could then inform the Client that this has happened by means of a high priority Message as follows

```
public void clientQueueThresholdReached(
 Client client,
 boolean upper,
 int threshold) throws APIException {

 if (upper) {
  TopicMessage message = Publishers.createDeltaMessage("MyTopic",50);
  message.put("Client Queue Reached Threshold of "+threshold);
  client.send(message,MessagePriority.HIGH);
 }
}
```

Low Priority

A message can be sent with low priority so that it will not interfere with the delivery of normal messages. Low priority messages will only be considered for delivery when there are no high or normal priority messages queued.

Acknowledged Messages

Client acknowledgement of Messages sent by a Publisher

Under normal circumstances, when a Message is sent to a Client by a Publisher it is queued for the Client and delivered via the normal queue and send mechanisms employed by Diffusion. The Publisher is not notified of delivery of the Message to the Client.

There is, however, the facility to force Client acknowledgement of Messages sent by a Publisher by using the setAckRequired method on a Message. This assigns an 'ACK Id' to the Message and when it is published (or sent) then all Clients that receive the Message must acknowledge it within a specified time period otherwise the Publisher will be notified of non acknowledgement. When the Publisher receives notification of non acknowledgement it will be supplied with the Message 'Ack Id' and a list of the Clients that did not acknowledge the Message.

The same facility exists from Client to Server. A Message sent from a Client may have 'ACK required' set, in which case the Server (or Publisher) must respond within a given time otherwise the Client application is notified of non acknowledgement of the Message

Message acknowledgement should be used sparingly (see considerations below).

Setting Messages as Requiring Acknowledgement

A message is set as 'requiring acknowledgement' using the setAckRequired method. This must be called before any user headers or data are added to the Message

For example

```
TopicMessage message = Publishers.createDeltaMessage("MyTopic",4);
message.setAckRequired();
message.put("Data");
publishMessage(message);
```

Copyright 2013 Push Technology

After setAckRequired has been called the Message's ACK Id can be obtained using the getAckId method.

Once a Message requiring acknowledgement has been published or sent it cannot be reused (sent again) - i.e. each Ack Id can only be used once. If it is required to set up such a Message and send several times then the message must be copied using the duplicate method which would assign a new AckId.

When a message sent from a Publisher is set as requiring acknowledgement it is automatically sent with high priority and will not be conflated.

Setting Acknowledgement Timeout

The time period within which a Message must be acknowledged may be configured for a Publisher in publishers.properties. A Publisher may programatically override the configured default timeout using the setAckTimeout method on the Publisher.

At the Client end the default ACK timeout can be set for the Client connection.

The timeout may also be explicitly specified for a Message using the setAckTimeout method on the Message itself.

Acknowledgement Timeout and Publisher Notification of Non Acknowledgement

When Diffusion™ receives a message from the publisher with the ACK flag set, a timer starts and Diffusion™ queues the message. If the client does not respond with an ACK before the timer elapses, or the message has not been queued before the ACK timeout expires, a NAK message is generated which notifies the publisher. It then becomes the publisher's responsibility to resend the message.

The message ack timeout clock runs from the time the message is sent to Diffusion™ by the message broker, as opposed to when it is physically put on the wire to a specific client. As a result, if a client is on the end of a slow network connection, then as the message queue continues to build, messages nearing the head of that queue may be nearing their timeout. When such a queued message reaches it's timeout, it is discarded and the publisher is informed via the generated NAK.

Client Handling of Acknowledgements

From the Client APIs point of view, Message acknowledgements can be handled automatically or manually.

When handled automatically, whenever a Message requiring acknowledgement is received from the Server then an ACK message is automatically sent back to the Server. This mode involves no special processing in the Client application.

When handled manually, when a Message requiring acknowledgement is received it is not automatically acknowledged and it will be up to the Client application to acknowledge it, possibly after processing it. There will be a method (isAckPending in Java) on Messages to identify those not acknowledged and a method on the connection (acknowledge in Java) to manually acknowledge the Message.

A Client connection has a method to set manual or automatic acknowledgement (setAutoAcknowledging in Java). The default is automatic.

Server Handling of Acknowledgements

How to handle message acknowledgements at the server

Users of the Publisher API can cause Message acknowledgements to be handled automatically or manually.

When handled automatically, then whenever a Message requiring acknowledgement is received from a Client then an ACK message is automatically sent back to the Client. This mode involves no special processing in the Publisher application.

When handled manually, then when a Message requiring acknowledgement is received it is not automatically acknowledged and it will be up to the Publisher to acknowledge it, possibly after processing it. The `isAckPending` method on a Message indicates whether it is in need of acknowledgement. Such a Message may be acknowledged within a Publisher via the `Client.acknowledge` method which sends an ACK notification back to the Client.

Copyright 2013 Push Technology

Whether a Publisher operates in automatic or manual acknowledgement mode is determined by the `auto-ack` property in *etc/Publishers.xml* .

Publisher Notification of Non Acknowledgement

A Publisher will receive notifications of non acknowledgement of Messages via its messageNotAcknowledged method.

For example

```
public void messageNotAcknowledged(
TopicMessage message,
List<TopicClient> clients) {
    LOG.warn("Message {} not acknowledged by:-",message);
    for (TopicClient client:clients) {
        LOG.warn("Client '{}'",client);
    }
}
```

Client Notification of Non Acknowledgement

A Client that sends Messages requiring acknowledgement to the Server must declare a listener for non acknowledgement notifications. In the Java API the connection has a setAckListener method for this purpose). Any attempt to send a Message requiring acknowledgement when no listener has been declared will fail.

Message Acknowledgement Considerations

Message Acknowledgement should be used sparingly and with care, taking the following into consideration

- A Message set as 'requiring acknowledgement' cannot be reused unless it is copied (using duplicate). Efficiencies of Message reuse (for example, using the same Topic Load message for many subscriptions if the data has not changed) would therefore be lost.
- A Message set as 'requiring acknowledgement' by a Publisher is always sent with high priority and thus the normal Client queueing mechanisms are by-passed. Queueing of expedited Messages is less efficient than normal queueing as such Messages have to be inserted at the head of the queue rather than being added at the end of it.
- Acknowledged Messages cannot be conflated.
- The handling of Acknowledged Messages involves extra processing at both the Client and at the Server therefore there will be performance implications.

Diagnostic Considerations

Messages requiring acknowledgement are actually a different Message Type (see Protocol) from normal Topic Messages so from a diagnostic point of view it should be borne in mind that the Message Type number will be different and there will be an additional (Ack Id) header.

Fragmented Messages

Sending messages in smaller chunks

Although most Diffusion messages are small, it is sometimes necessary to send messages larger than the normal limits. Fragmentation allows a Publisher to send a message in smaller chunks over a period of time, and for that message to be reconstituted by the client. Advantages of this technique are

- Message size does not need to be constrained by the configured maximum message size
- Other messages may be interleaved, so that the sending of one very large message does not block subsequent messages.

Copyright 2013 Push Technology

Creating fragmented messages

Fragmented messages can be created by setting the fragmentation settings of a Topic Message. The message will be fragmented only if it is larger then the fragment size.

Table 42: Creating a Fragmented Message

`TopicMessage message = createLoadMessage(size); message.setFragmentLifeCycle(new FragmentedMessageLifecycle(delay)); message.setFragmentSize(fragmentSize);`	Creates a new, empty Initial Load TopicMessage with a fragment lifecycle and a fragment size.
`TopicMessage message = createDeltaMessage(size); message.setFragmentLifeCycle(new FragmentedMessageLifecycle(delay)); message.setFragmentSize(fragmentSize);`	Creates a new, empty Delta TopicMessage with a fragment lifecycle and a fragment size.

The size parameter is the maximum size of the message (excluding headers).

The lifecycle is an object which provides control over how the message fragments are queued over time:

Table 43: Fragmented Message Lifecycle

`FragmentedMessageLifecycle(int sendingDelay);`	Specify a delay (in milliseconds) between sending each fragment of a message.
`FragmentedMessageLifecycle(int sendingDelay, long maximumAge);`	Also set the maximum amount of time (in milliseconds) that is allowed to pass after the first message is queued before we consider the message to be too old to be of any use to the clients. After this time has elapsed, a cancellation message is sent to subscribed clients.
`FragmentedMessageLifecycle(int sendingDelay, long maximumAge, long priorityBumpTime);`	As above, but if the messages are still sending after priorityBumpTime milliseconds (which should be less than the maximum age) then the priority of remaining fragments should be increased.

The fragmentSize parameter is the size of the fragments the message will be broken into.

Notes

- If the sendingDelay is 0 (or less) then there will be no delay between sending each fragment and no other fragments or messages will have the opportunity to be interleaved with this message.
- The FragmentedMessageLifecycle class has accessor methods for manipulating the above values to provide a different level of control, e.g. setting the priority bump time while allowing a non-expiring maximum age for the fragments.
- A size of -1 turns off fragmentation for the message, and a size smaller than 4 is illegal. Although very small fragment sizes are allowed, their use is discouraged. Typically, you will request a fragment size as large as is realistically possible.

Copyright 2013 Push Technology

- A `MessageException` will be thrown if you try to set the fragment size larger than the maximum message size.
- The fragment size can be set larger than the inital size of the message as a message can be resized if it exceeds the inital size.

Message Filters

Accepting, rejecting or replacing Messages

Message Filters are a feature within the Publisher API which provide the ability to accept, reject or replace a Message being processed. It is a user written class which must implement the MessageFilter interface which has a single selectMessage method which is passed a TopicMessage and can return:

- The same Message as passed in - Thus 'accepting" the Message.
- Null - Thus 'rejecting' the Message. i.e. indicate that it should not be processed.
- A different Message from that passed in - Thus replacing the Message.

Replacement would need to be done with care as it would involve creating a new Message which should have the same basic characteristics of the original (topic name etc).

Message Filters are used to filter what is queued for a particular Client - see below.

Client Queue Message Filtering

It is possible to specify Message Filters to be used for a specific Client to filter the Messages that get queued for the Client.

Messages get queued for a Client as a result of a publish to all Clients subscribed to a Topic or an explicit 'send' to the Client. However, it may be that you do not want all Messages published on a Topic to go to a particular Client instance.

For example, you may want to only send a Message to a Client if a value in the Message is greater than a certain amount or you may want to send messages via some timed 'tick' rate (similar to throttling).

Such a filter can be set for a Client to handle Messages for a specified Topic. It is also possible to set a filter for a Topic Selector pattern so that it would be used for any Message with a Topic that matches the pattern.

A filter may be set using the addQueueMessageFilter method on the Client interface. Filters may be removed using removeQueueMessageFilter. See the javadoc for more details.

Filter Example

The following example shows a Publisher filtering Mobile Clients so that they only receive Messages with the first user header set to "M"

```
@Override
protected void subscription(Client client,Topic topic,boolean loaded)
throws APIException {

 if (client.getConnectionType().isCategory(ConnectionCategory.MOBILE)) {
  client.addQueueMessageFilter(topic.getName(),new MobileFilter());
 }

 super.subscription(client,topic,loaded);
}

private class MobileFilter implements MessageFilter {
 @Override
 public TopicMessage selectMessage(TopicMessage message) {
  if ("M".equals(message.getHeader(0))) {
```

Copyright 2013 Push Technology

```
    return null;
   }
   else {
    return message;
   }
  }
 }
```

The above example shows a filter being set for every Topic which may not be very efficient

Single Filter for all Topics

The same could be achieved by setting a single filter for all Topics when the Client connects, for example

```
/**
 * @see ClientListener#clientConnected(Client)
 */
@Override
public void clientConnected(Client client) {
 if (client.getConnectionType().isCategory(ConnectionCategory.MOBILE)) {
  try {
   client.addQueueMessageFilter("//",new MobileFilter());
  }
  catch (TopicInvalidException ex) {
  }
 }
}
```

Care must be taken though when filters are applied in this way to ensure that you are not accidentally filtering Topics that should not be filtered.

Copyright 2013 Push Technology

Copyright 2013 Push Technology

Chapter 12

Security

Topics:

- *User Access Control*
- *Network Security*

This section covers security issues within Diffusion™ and describes the facilities available within Diffusion™ relating to security.

User Access Control

Diffusion™ provides facilities for controlling the access of Users to a Diffusion™ Server.

The access of Clients may be controlled using Client Validation policies and also by the use of an Authorisation Handler.

An Authorisation Handler can control authentication, authorisation and permissioning.

Authorisation Handler

An Authorisation Handler is a user written Java class that must implement the AuthorisationHandler interface in the PublisherAPI.

Such a handler may be used to restrict access of Clients according to any criteria that is appropriate. One capability within Diffusion™ is for a Client to be able to specify 'Credentials' when they connect that may then be checked by the Authorisation Handler.

The handler may either be specified in *etc/Server.xml* in which case it is loaded when the Server starts or may be set programatically within a Publisher using the `Publishers.setAuthorisationHandler` method.

There can only be one handler and it is system wide across all publishers, although it is possible to have authorisation at the publisher level.

If an authorisation handler is not specified, then credentials sent by a `Client` are assumed to be valid. A `Publisher` will have access to the `Credentials` to perform finer-grained authorisation, if required.

The authorisation handler interface has the following methods:-

Table 44: Authorisation Handler Methods

`canConnect(Client)`	This method is called to establish whether the Client can connect and is called before any Client Validation Policy is called.
`canSubscribe(Client, Topic)`	This method is called when a Client subscribes to a Topic.If Topic information is sent with the connection, then this method is called after the `canConnect` method. Note that this will be called for every Topic being subscribed to,even if subscribed as a result of a Topic selector being specified. However (by default),if a Topic is rejected by this method then it will not be called again for any children (or descendants) of the Topic.
`canSubscribe(Client,TopicSelector)`	This method is called when a Client attempts to subscribe to a Topic Selector pattern (as opposed to a simple Topic name). If Topic information is sent with the connection, then this method is called after the `canConnect` method.
`canFetch(Client,Topic)`	This method is called when a Client sends a `fetch` request to obtain the current state of a Topic. Note that this will be called for every Topic being fetched, even if fetched as a result of a Topic selector being specified. However (by default), if a Topic is rejected by this method then it will not be called again for any children (or descendants) of the Topic.

Copyright 2013 Push Technology

`canFetch(Client,TopicSelector)`	This method is called when a Client attempts to fetch Topics using a Topic selector pattern (as opposed to a simple Topic name).
`canWrite(Client,Topic)`	This method is called when a Client sends a Message on a given Topic, if false is returned the message is ignored, and the Publisher will not be notified of the Message.
`credentialsSupplied(Client,Credentials)`	This method is called when a Client submits Credentials after connection. It may be used to validate the credentials and must return true if the Credentials are OK. If this returns false then a 'Credentials Rejected' message will be sent back to the Client. This method should set the Credentials on the Client if they are found to be OK.

Authentication

When a Client connects to Diffusion™ it has the option of supplying user credentials. These credentials are basically tokens called username and password. These tokens can be used for any purpose. When `canConnent` is called, it is possible to get the Credentials from the Client Object.

An example of this is:-

```
   public boolean canConnect(Client client) {
Credentials creds = client.getCredentials();

// No creds supplied, so reject the connection
if (creds == null) {
    return false;
}

String username = creds.getUsername().toLowerCase();
```

If the credentials are null, then none were supplied, which is different from empty credentials.

Clients may connect without Credentials and submit them later or replace the Credentials at any time whilst connected. The Authorisation Handler will be notified when new Credentials are submitted and may choose to set the new Credentials on the Client.

The `Credentials` class has username and password attributes, but also allows for an attachment. It is here that a user would normally set any security object required. Returning true will allow the user to connect, returning false will result in the Client connection being refused.

Authorisation

Authorisation is the allowing of a Client to subscribe to a Topic. In this case the `canSubscribe` is called. Returning true here will allow the Publisher to have any Topic Loaders and subscription methods called. Returning false will not notify the Client that the subscription was invalid. A good design pattern here is to look at the object attached to the clients's credentials to see what authorisation by Topic is required.

```
public boolean canSubscribe(Client client, Topic topic) {

     // Everyone has access to the top level topic
     if (topic.getName().equals(CHAT_TOPIC)) {
```

Copyright 2013 Push Technology

```
     return true;
 }

 User user = (User) client.getCredentials().attachment();

 return user.isAllowedRoom(topic.getNodeName());
}
```

Authorisation Handler

Authorisation at the Publisher level can also be achieved. This is required if there are many Publishers running within the same Diffusion™ Server and they have different security settings. The code snippet below will work if the Publishers all implement `AuthorisationHandler`

```
public boolean canSubscribe(Client client, Topic topic) {

    AuthorisationHandler handler =
             (AuthorisationHandler)Publishers.getPublisherForTopic(topic);

 // Call the publisher in question
 return handler.canSubscribe(client, topic);
}
```

Permissioning

The permissioning process governs weather a client is able to send Messages to a Publisher, or in other words, is the Topic read only. This is handled by the `canWrite` method. Again a good pattern may be to look at the Credentials attachment object to see if this is permissible.

```
 public boolean canWrite(Client client, Topic topic) {
 User user = (User) client.getClientCredentials().attachment();
 return user.canWriteMessages();
  }
```

Network Security

This section describes how to deploy network security, which can be used in conjunction with data security.

Secure Clients

Below is a table of the clients and whether they can connect securely via SSL and/or HTTPS.

Copyright 2013 Push Technology

Table 45: Client Security

Client	DPTS	HTTPS
.Net	✔	✔
Java	✔	✔
Flash	✘	✔
Silverlight	✘	✔
WebSocket	✔	✘
XHR	✘	✔
Android	✔	✘
iOS	✔	✘
J2ME	✔	✘

It is also possible to have Diffusion™ server master/slave connections over DPTS as well.

Connector Configuration

If secure connections are required, Diffusion™ connectors need to be configured to support DPTS and/or HTTPS. Any connector can accept DPTS connections. A connector need not be dedicated to only DPTS connections. DPTS connections can also accept clear text connections. To enable DPTS connections the connector configuration will need a keystore entry. This informs the connector that it is enabled for DPTS type connections. If HTTPS is required, a keystore section and a web-server entry also will be required, even for secure Diffusion™ clients.

Keystores

When connecting, an incorrect Keystore will result in SSL errors. One of these being no cipher suites in common. Below are examples of how to create Keystores. This will vary from provider to provider.

With Verisign Certificate

```
   # Generate Key (place in keystore (keystore created if it does not
exist))
#  - website domain name to secure, for example: demo.pushtechnology.com
#  - depends on SSL provider
keytool -genkeypair -alias  -keyalg RSA -keystore  -keysize

# Generate CSR Request for SSL provider
keytool -certreq -keyalg RSA -alias  -file certreq.csr -keystore

# Receive certificate back from SSL Provider.

# for Verisign SSL certificates, you need to install the primary and
secondary intermediate certificates before your own:
```

Copyright 2013 Push Technology

```
keytool -import -trustcacerts -alias primaryIntermediate -keystore  -file
primary_intermediate.crt
keytool -import -trustcacerts -alias secondaryIntermediate -keystore  -file
secondary_intermediate.crt

# Then install your own, note to use the same alias as the CSR request and
key
keytool -import -trustcacerts -alias  -keystore  -file
certfromsslprovider.crt

#This should install your Cert and chain it together with the CA providers
primary and secondary certs.
#check by:
keytool -list -keystore  -v
Keystore type: JKS
Keystore provider: SUN
Your keystore contains 3 entries
Alias name:
Creation date: Aug 26, 2011
Entry type: PrivateKeyEntry
Certificate chain length: 4
```

With goDaddy Certificate

```
keytool -genkeypair -alias  -keyalg RSA -keystore  -keysize

# Generate CSR Request for SSL provider
keytool -certreq -keyalg RSA -alias  -file certreq.csr -keystore

# Receive Cert back from SSL Provider.
# Download the Tomcat pack and get the root certs from https://
certs.godaddy.com/anonymous/repository.pki
keytool -import -alias root -keystore keystore -trustcacerts -file
valicert_class2_root.crt
keytool -import -alias cross -keystore keystore -trustcacerts -file
gd_cross_intermediate.crt
keytool -import -alias intermed  -keystore keystore -trustcacerts -file
gd_intermediate.crt
keytool -import -alias demo.pushtechnology.com  -keystore keystore -
trustcacerts -file demo.pushtechnology.com.crt
```

Copyright 2013 Push Technology

Chapter 13

Distribution

Topics:

- *Publisher Clients*
- *Distributed Topics*
- *Distribution Examples*

In many cases all that is required of a Diffusion™ implementation is a single Diffusion™ Server hosting one or more Publishers. However, Diffusion™ provides an infinitely scalable distributed solution which can involve any number of Diffusion™ Servers and any number of Publishers.

The following sections discuss distribution in more detail:-

Publisher Clients

What is a Publisher Client?

A Publisher can connect 'up stream' to a Diffusion™ Server as if it were a Client connecting to that Server and can then interact with the server like any other client. When a Publisher acts in this way it is said to be acting as a 'Publisher Client'.

Publisher Client Capabilities

From within the Publisher a Publisher Client connection to a Server is represented by the `PublisherServerConnection` interface. This is a subtype of `ServerConnection` which is the same interface used by connections to Servers from External Clients. A Publisher acting as a Client therefore has all the same capabilities as any External Client. For example, it can subscribe to Topics on that Server or can send Messages to Topics on that Server.

For full details of the capabilities see the issued Javadoc for the ServerConnection interface.

Publisher Client Connection

Connecting a Publisher to another Diffusion Server

A Publisher connects to a Server listening on a given port at a specified host. The host and port will correspond to a Client Connector configured at that Server.

Connecting using API

A Publisher may register a connection to a Server using `addServerConnection` and then use the `ServerConnection.connect` method to make a connection. Within a Publisher the following code creates a new server connection to the 'localhost' on port 8080:

```
PublisherServerConnection conn =
 this.addServerConnection("ServerLocalName","localhost",8080);
```

This connection is named 'ServerLocalName', this is the name that the Publisher uses to refer to the connection. The `addServerConnection` method can take a ServerDetails object instead of the host and port.

```
PublisherServerConnection conn =
 this.addServerConnection("ServerLocalName",ConnectionFactory.createServerDetails("dpt:/
localhost:8080"));
```

You should refer to the Javadoc for more information on how to use `ConnectionFactory.createServerDetails`.

A failure policy may be set for the connection before connecting which defines the action to take if the connection fails or is lost at some point after connection.

The connection is not opened as soon as it is declared, it must be opened explicitly with `connect`. Once the connection has been opened then the connection can be used to subscribe to topics.

When connected the Publisher will be notified via the `serverConnected` method and the Publisher would then be able to subscribe to Topics on the Server as required.

A complete example of how to connect using the API.

```
PublisherServerConnection conn =
 this.addServerConnection("ServerLocalName","localhost",8080);
conn.setFailPolicy(PublisherServerConnectionFailPolicy.RETRY);
try {
 String clientId = conn.connect();
}
```

Copyright 2013 Push Technology

```
catch (APIException e) {
 ...
}
```

Configured Connection

It may be more convenient to configure the Servers that a Publisher should automatically connect to in the *etc/Publishers.xml* file. In this case any Servers that are declared and connected before `initialLoad` is called, and all connections that have not been explicitly connected to will be automatically connected after `initialLoad`.

A failure policy may be configured so that if a connection fails it will either stop the Publisher or attempt to periodically retry the connection. The default policy is to ignore a failed connection, thus allowing the Publisher to deal with it.

Any number of server connections can be configured as elements inside the publisher element of the *etc/Publishers.xml* . The following is a minimal example, all elements of server in it are required. You should refer to the XSD for further information.

```
<publisher name="ChildPublisher">
 ...
 <server name="ServerLocalName">
  <host>localhost</host>
  <port>8081</port>
  <input-buffer-size>4k</input-buffer-size>
  <output-buffer-size>64k</output-buffer-size>
  <fail-policy>RETRY</fail-policy>
 </server>
</publisher>
```

Connection Failure Policy

A connection failure policy may be defined for a Server connection to indicate what should happen if the connection is lost at some point after the connection is made. For configured connections the failure policy would also apply to failure to connect during startup.

A failure policy may be set at any time on a connection using the `setFailPolicy` method.

For configured connections the failure policy may be specified in *etc/Publishers.xml*

Policy options are as follows:-

DEFAULT	For a configured connection, failure to connect at startup is logged and the Publisher could determine the state of such connections after `initialLoad` (e.g. in `publisherStarted`) using the `ServerConnection.getState` method. If a connection is lost, the Publisher is notified via the `serverDisconnected` method but no further action is taken.
CLOSE	For a configured connection, failure to connect at startup causes the Publisher to be stopped. For any type of connection if the Publisher loses its connection to the Server then the Publisher will be closed.
RETRY	For a configured connection that fails during startup or any connection that is lost after connection is made

Copyright 2013 Push Technology

	then the connection will be periodically retried until it is reconnected. The default retry interval is 5 seconds but this can be specified if required.

Using the Connection

Managing a Publisher's server connections is done using the name of the connection to identifiy it.

Any named connection can be retrived, regardless of whether it was created using the API or the configuration.

```
PublisherServerConnection conn =
 this.getServerConnection("ServerLocalName");
```

Any named connection can be closed and removed from the Publisher using `removeServerConnection`.

The Publisher can test to see if a name has been used to describe a Server Connection using `hasServerConnection`.

When a Publisher connects to a Server the `serverConnected` method is called. This uses the ServerConnection interface to pass the connection information. This method can be overriden so that you can implement custom behaviour when a server is connected. Once connected to the server it is possible to interact with it's Topics, it can subscribe and send messages.

```
@Override
protected void serverConnected(ServerConnection conn) {
 if (conn.getName().equals("ServerLocalName")) {
  conn.subscribe("TopicName");
  ...
 }
}
```

The `serverDisconnected` method is called to notify the Publisher a server has disconnected. This uses the ServerConnection interface to pass the connection information. This method can be overriden so that you can implement custom behaviour when a server is connected.

```
@Override
protected void serverDisconnected(ServerConnection conn) {
 if (conn.getName().equals("ServerLocalName")) {
  ...
 }
}
```

The Publisher is responsible for processing messages that are received from the server using the method. This uses the ServerConnection interface to pass the connection information.

```
@Override
protected void messageFromServer(ServerConnection conn, TopicMessage
 message) {
...
}
```

The ServerConnection can be used to construct and send messages to the server. It provides `createLoadMessage`, `createDeltaMessage` and `send` to communicate with the server. The server treats a Publisher Client as any other client and no special handling is required on its side.

Notifications

A Publisher is notified of a connection to a Server via its `serverConnected` method. This is therefore the ideal place to subscribe to Topics on the Server.

Copyright 2013 Push Technology

Once connected a Publisher receives Messages from Servers on its `messageFromServer` method which indicates from which Server the Message came.

A Publisher will receive notification of the change of status (e.g removal) of any Topics it subscribed to on a Server via the `serverTopicStatusChanged` method.

The Publisher is notified of a lost connection on it's `serverDisconnected` method.

Buffer Sizes

For a configured connection the sizes of the input and output buffers to use for the socket connection may be specified in *`etc/Publishers.xml`*

The buffer sizes may also be specified (or changed) for a `ServerConnection` by obtaining the `ServerDetails` (using `getServerDetails`) and using `setInputBufferSize` and setOutputBufferSize. These methods must be called before connect is called.

It is important that the buffer sizes specified are compatible with those at the other end of the connection. Remember that the input buffer should match the Server's output buffer and the output buffer should match the Server's input buffer.

Distributed Topics

How to distribute Topics across Servers

Within Diffusion™, Topics are unique within any one Server but there is no restriction on serving more than one Server in a distributed environment having Topics of the same name. Indeed, this is often required.

Diffusion™ itself does not impose any synchronisation of Topics across Servers. This is to allow maximum flexibility of the configuration of Topics. There is no reason why a Publisher that subscribes (as a Client) to the Topic(s) of a Publisher at another Server needs to replicate those Topics in its own Server. However, in some circumstances (e.g distributors) this may be exactly what is required.

The behaviour of distributed Publishers is therefore in contrast to Event Publishers, where Messages for a Topic are routed to the Publisher that owns the Topic.

For example, a Publisher may subscribe to a Topic called 'Feeder' on another Server but that does not mean that there has to be a Topic called 'Feeder' on its own. The Publisher may choose to transform the data received on the 'Feeder' Topic and publish it on an entirely different Topic.

Topic Replication

Where Topic replication is required from the server that is connected to this can be greatly simplified by using *`TopicNotifyTopicData`* in the master server and then the slave server can subscribe to the notifying Topic and receive notifications of Topics added.

It is not necessary to replicate all Topics at the master server as the notification facility can be used to notify some Topic creations.

In the case where replication is required then full notifications must be requested. The `TopicDefinition` objects passed in the notifications can be used directly to create duplicate Topics local to the Publisher Client.

Note that Remote Control Topics (or their subdomain Topics) cannot be replicated by this mechanism as the creation of such Topics from a TopicDefinition is not supported. A Remote Control Topic could be created from the information in a definition but all Remote Service and subdomain Topics exist in the context of a Remote Service connection and cannot be replicated in this way.

Distribution Examples

The following are some Distribution scenarios within Diffusion™:-

Copyright 2013 Push Technology

Distributors

Data Distribution

This is the term used to represent the situation where there is a single Publisher publishing messages to one or more other Publishers in other Diffusion™ Servers.

In this case the one that distributes the messages is known as the 'distributor' Publisher and the others are 'recipients'. The 'recipient' Publishers connect to the Server of the 'distributor' publisher and subscribe to its Topics as Publisher Clients. Messages that the distributor publishes are then published to all Clients which in this case are Publishers.

This scenario could be used to balance client connections across a number of Diffusion™ Servers all serving the same data.

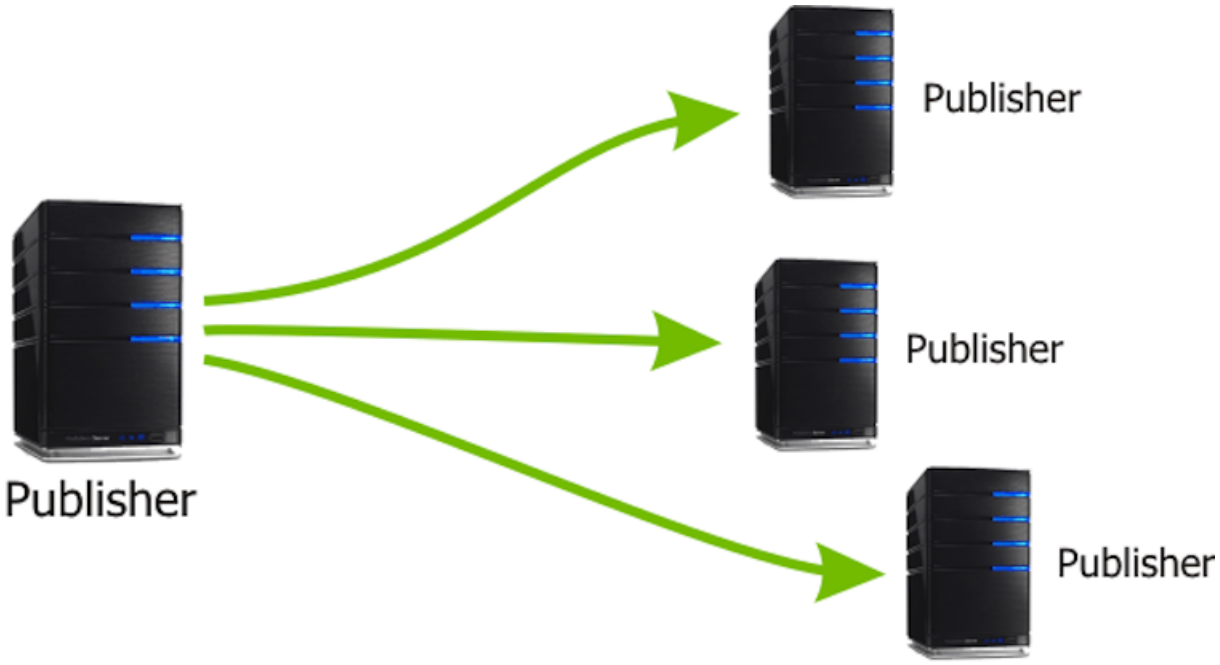


Figure 25: Distributors

Aggregators

Data Aggregation

This is the term used to describe the situation where a single Publisher aggregates messages received from one or more other Publishers.

In this case the 'aggregator' connects to more than one other Diffusion™ Server and subscribes to Topics on them.

This scenario could be used to aggregate data being received from more than one source providing a single point of contact for Clients which see only the aggregated data.

Copyright 2013 Push Technology

Figure 26: Aggregators

Mixed Mode

The client/server relationship between Publishers is not restricted to the above examples and a peer to peer network of Publishers could be set up which communicate with each other in any way desired.

A Publisher publishes to Clients and there is no reason why the Clients connected to a Diffusion™ Server could not be a mixture of Publisher Clients and other Clients.

Copyright 2013 Push Technology

Copyright 2013 Push Technology

Chapter 14

Event Publishers

Topics:

- *Receiving Messages from Event Publishers*
- *Receiving Event Publisher Notifications*
- *Sending Messages to Event Publishers*
- *Event Publishers Receiving Messages from Publishers*
- *Configuring Event Publishers*

An 'Event Publisher' is an application that is able to connect to a Diffusion™ Server and publish Messages to a Topic (or Topics). These Messages are then be routed to the Publishers that own those Topics.

The purpose of an Event Publisher is to feed data into a Diffusion™ Server and thus into Publishers.

Publishers may also send messages to Event Publishers but high volumes in this direction would not be expected

An Event Publisher may be written using the Java API or the Windows API. An Event Publisher could also be written in any other language that can communicate over TCP sockets via DPT.

Receiving Messages from Event Publishers

A Publisher will receive Messages from Event Publishers via its `messageFromEventPublisher` method. The Message (a `TopicMessage`) and a reference to the Event Publisher Connection (`EventConnection`) are passed. It is possible (but not typical) for a Publisher to receive Messages from more than one Event Publisher connection.

Receiving Event Publisher Notifications

It is also possible to receive notifications when an Event Publisher connects or disconnects. This may be important as the Publisher may be relying upon the Event Publisher for the integrity of its data. To receive such notifications the Publisher adds a listener using the `Publishers.addEventListener` method.

Sending Messages to Event Publishers

Messages may be sent from Publishers to Event Publishers using the send method on the `EventConnection` interface. Only delta Messages may be sent.

Event Publishers Receiving Messages from Publishers

Event Publisher applications will receive Messages from Publishers on their listener interface.

For example, in the Java API, such a Message is received on the `EventPublisherListener.messageFromServer` method.

Configuring Event Publishers

Event Publishers are configured programmatically by setting values of the Event Publisher Connection.

Each Event Publisher has one or more `EventPublisherConnection` objects that it uses to communicate with a Publisher. Each of these connections can be configured separately. An `EventPublisherConnection` is configured programmatically through the Java API.

There are two objects that can be configured for the Event Publisher, the `EventPublisherConnection` and the `ServerDetails`.

The `EventPublisherConnection` object allows the connection host, port, message queue size, topic aliasing and a listener to be configured.

- The host and port are used to identify the Diffusion™ server, these can also be configured in an `ServerDetails` object.
- The message queue size is the number of messages that can be queued on the Event Publisher side to be sent to the Publisher.
- Topic aliasing can be enabled or disabled for use with the Event Publisher.
- The listener allows the Event Publisher to handle disconnections and messages from the Publisher.

The `ServerDetails` object allows the connection host, port, auto-acknowledgements, connection timeout, credentials, input and output buffer sizes, proxy, SSL context, topics and a write timeout to be configured.

- The host and port are used to identify the Diffusion™ server, these can also be configured in an `EventPublisherConnection` object.
- The auto-acknowledgements allows messages that require acknowledgement to be acknowledged automatically.

Copyright 2013 Push Technology

- The connection timeout refers to how much time will pass before the Event Publisher accepts that it cannot open a connection.
- The credentials that are used to authenticate a connection can be configured.
- The input and output buffer sizes are the size of the read/write socket buffers.
- The proxy allows connections to go through a proxy server.
- The SSL context is used to set up secure connections.
- The topics that will be subscribed to automatically on connection.
- The write timeout is how much time will pass before the Event Publisher accepts that it cannot write to the connection.

Further details of the `EventPublisherConnection` and the `ServerDetails` can be found in the Javdoc for these classes.

Copyright 2013 Push Technology

Copyright 2013 Push Technology

Chapter 15

Remote Control

Topics:

- *Remote Control*
- *Remote Services*
- *Remote Control Publishers*

Remote Control allows applications using the Remote Service API to communicate with Remote Control enabled Publishers in a Diffusion™ Server.

Remote Control

Remote Control allows applications using the Remote Service API to communicate with Remote Control enabled Publishers in a Diffusion™ Server.

A Remote Service application can connect to a Diffusion™ Server and perform many of the functions that could be achieved within a Publisher, such as creating Topics and subscribing Clients to them. This allows application architecture to be split into two tiers - the'edge' which interfaces with Clients and the 'back end' or 'remote' tier which provides 'service' functionality.

Remote Control functionality centres on the concept of a Remote'Control Topic' at the Server with which remote applications are able to communicate, in order to register 'services'. A 'service' is represented at the Server by its own Topic which is referred to as 'a Service Topic' or a 'Domain Topic'. The Remote Service then has the ability to create new Topics subordinate to the Service Topic which are known as 'Subdomain Topics'. These Subdomain Topics may be of various types depending upon their intended functionality. For example, 'Publishing Topics'can be created which allow messages to be sent from the Remote Service for publishing to all subscribed clients.

The concept of a 'Domain' means that a Remote Service can only manipulate Topics within its Domain (i.e. subordinate to the ServiceTopic). It also means that all requests from Clients relating toTopics within the Domain will be routed to the Remote Service rather than performed within the Server. For example, when a Client subscribes to a subdomain Topic it will not immediately be subscribed but a request will be sent to the Remote Service indicating that the Client wishes to subscribe. The Remote Service may then authenticate the subscription and if allowed send a subscribe request back to the Server which will perform the actual subscription of the Client to the Topic.

Two-Tier Architecture

The simplest form of Remote Control might be used in a two tier architecture where the back end services communicate directly with a Diffusion™ 'edge' server.

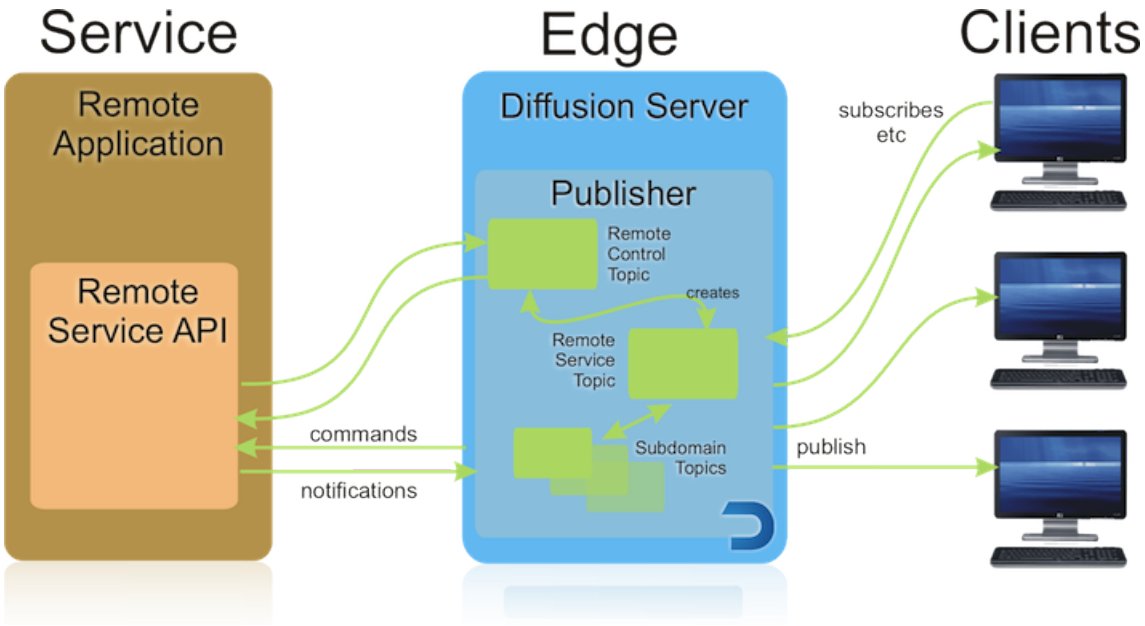


Figure 27: Two-Tier architecture

In this case the Remote Service sends requests directly to the edge Publisher to be executed and all domain specific Client requests are routed to the corresponding Remote Service for processing.

Three-Tier Architecture

A more complex form of Remote Control can be deployed in a three tier architecture where the Remote Service communicates with aPublisher in a 'relay' or 'routing' tier which in turn communicates with an edge tier server.

Copyright 2013 Push Technology

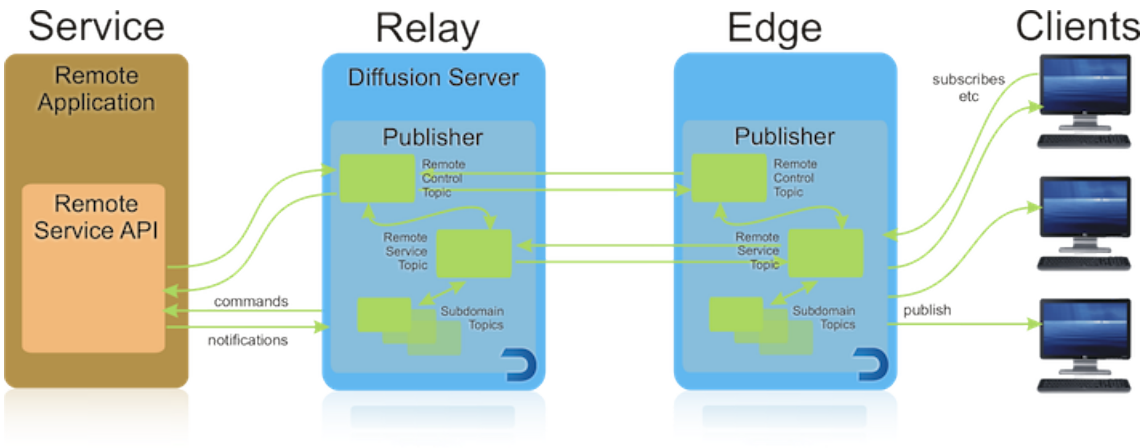


Figure 28: Three Tier Architecture

The relay tier forwards requests on to a Diffusion™ edge server. Client requests at the edge are routed via the relay tier back to the Remote Service. The domain Topic tree is maintained in both the edge and relay servers though the Topics in the relay tier only reflect the Topic structure at the edge and do not maintain state. All Client connection information at the edge is notified to the relay server which maintains Client maps for the benefit of RemoteServices.

Remote Services

A Remote Service is an application using the Remote Service API to communicate with a Remote Control enabled Publisher in a Diffusion™ Server.

Remote Service API

The Remote Service API enables applications to communicate with a Publisher within a Diffusion™ Server that implements Remote Control such that the Remote Service application can perform much of the processing that would normally be performed within a Publisher.

The API is available in both Java and .NET and connects to the server (relay or edge) over an Event Publisher connection. See the *Java Remote Service API* or the .NET Remote Service API section for more details of how to use the API.

Remote Service Capabilities

The Remote Control facility allows an application to be able to connect to a Diffusion™ Server and benefit from the following capabilities:-

Service Registration

An application that wishes to act as a Remote Service can connect to a Diffusion™ Server and register itself as a Remote Service. In order to do this it must specify the following:-

- **Server Host and Port**

  In order to locate a running instance of a Diffusion™ Server.
- **Control Topic Name**

  The name of a Topic that already exists at the specified Server and has been set up as a Remote Control Topic.
- **Domain Topic Name**

  The name of a Topic that is to be created at the specified Server on behalf of the Remote Service under which all Topics to be managed by the Remote Service are to be created. This Topic must not already exist at the Server.

Copyright 2013 Push Technology

The Remote Service will receive notification that it has been registered (or registration has failed). As soon as a Remote Service is registered then all requests from Clients for Topics subordinate to the domain Topic will be routed to the Remote Service (even if those Topics do not exist) and the Remote Service will have the ability to create and maintain Topics at the Server. The Remote Service will also receive notifications of Clients connecting (or already connected) as required.

Client Notifications

By default a Remote Service is only notified of the connection of a Client just before the first request from that Client is directed to the Remote Service. In other words, a Remote Service will not know of a Client unless the Client accesses a Topic which is subordinate to the service's Domain Topic. This lazy form of notification ensures that the Remote Service is not flooded with details of Clients that are not relevant to it.

However, there may be situations where a Remote Service wishes to be informed of all Client connections and this may be specified as an option before registering the Remote Service. In this case if there are Clients already connected then the Remote Service will be notified of all of them immediately after registration. If a Client changes its credentials after connection then (assuming the Remote Service has already been notified of the Client) the Remote Service will receive a notification of an update to the Client details.

The Remote Service API will maintain a map of all of the Clients known to it.

Topic Creation and Removal

A remote service can add and remove Topics subordinate to its Domain Topic. These Topics may then be used in various ways (depending upon Topic type) by the remote service and Clients that access such Topics will have their requests touted to the remote service.

The types of Topics that can be created are:-

Table 46: Topic Types

Type	Description
Simple	This is a Topic that will have no Topic Data. The Remote Service can publish messages to such a Topic as well as send messages to specific clients using it. Clients may send messages to this Topic and they will be routed to the Remote Service.
Single Value	This is a Topic that will use *Single Value Topic Data*. The Remote Service can publish messages to such a Topic.
Record	This is a Topic that will use *Record Topic Data*. The Remote Service can publish messages to such a Topic.
Protocol Buffers	This is a Topic that will use *Protocol Buffers Topic Data*. The Remote Service can publish messages to such a Topic.
Routing	This is a Topic that will use *Routing Topic Data*. The Remote Service will be able to subscribe Clients to such a Topic with actual target Topic as either a Single Value or Record Topic in the service Domain.
Service	This is a Topic that will use *Service Topic Data* but all requests to the service will be routed to the Remote Service for processing.
Custom	This is a Topic that will use *Custom Topic Data*. When specifying the Topic the full name of a Java class (available at the server) that implements `CustomTopicDataHandler` must be specified. The Remote Service can publish messages to such a Topic.
Paged	This is a Topic that will use *Paged Topic Data*. There are different subtypes of Paged Topic Data and the Topic Specification will provide a mechanism for creating a helper object to update the paged data in a manner appropriate to its type.

Copyright 2013 Push Technology

Type	Description
Slave	This is a Topic will use *Slave Topic Data*. It points at another (Publishing) topic (the 'master' topic). The master topic is specified when the Topic is created and can be any Topic, even one outside the service domain.
Child List	This is a Topic that will use *Child List Topic Data* to maintain a list of its child topics.
Topic Notify	This is a Topic that will use *Topic Notify Topic Data* to provide Topic add notification capabilities to clients.

If a request to create a Topic fails then the Remote Service will be notified.

Any Topic subordinate to the service domain can be removed by the Remote Service at any time. This will automatically cause the unsubscription of all Clients subscribed to the Topic. Note that removing a Topic also removes any subordinate Topics.

Subscription

When a Client attempts to subscribe to a subdomain Topic it will not be subscribed immediately but the request to subscribe will be routed to the relevant Remote Service. This allows the Remote Service to authenticate the Client and then explicitly subscribe it as required.

When a Client subscribe request is for a Routing Topic then the Remote Service would subscribe the Client specifying the desired target Topic for the Client. The service may need to create the target Topic dynamically first.

If a Client attempts to subscribe to a Topic within the domain that does not exist then the request would also be routed to the Remote Service. This gives the Remote Service the opportunity to create the Topic dynamically.

A Remote Service may subscribe a Client to any of it's subdomain Topics at any time, not only in response to a subscription request.

When subscribing to a Routing Topic then both the Topic name and the name of the Topic to route to need to be supplied.

A Topic selector may be supplied to subscribe many topics at once.

The edge processing will deal with the possibility that a Client has unsubscribed from a Topic since it originally issued the subscribe request and discard the subscribe.

Publishing

A Remote Service may publish messages to subdomain Topics resulting in the message being published to all subscribed Clients.

If the Topic is a Simple Topic then the message will be published to all subscribed Clients. If the Topic has Publishing Topic Data then an "update and publish" action will be performed which would publish if the data had changed. For all other Topic types a published message will be ignored.

Service Request Processing

If a Remote Service creates a subdomain Topic that has Service Topic Data then all service requests from Clients on the Topic will be routed to the remote service which should return a response.

Client Message Processing

On simple subdomain Topics (i.e. those without Topic Data) Clients may send messages to the Topic (which will be routed to the Remote Service) and the Remote Service may send messages to Clients.

Fetch Processing

If a Client attempts a 'fetch" on a subdomain Topic then the fetch request is routed to the Remote Service which can validate that the Client is allowed to do the fetch.

A Client can request a fetch on a Topic that does not exist but if the requested Topic name is in a known Domain then the request will be routed to the corresponding Remote Service.

Copyright 2013 Push Technology

On receiving a fetch request the Remote Service may return a fetch reply to the Client. There are two options for handling a fetch reply:-

1. The Remote Service may send a message that will be sent on to the Client as a the fetch reply.
2. The Remote Service may indicate that the current state of the Topic at the edge should be returned to the Client as a fetch reply. If the Topic has Topic Data then the state will be fetched from it otherwise the Publisher will be invoked to provide the fetch reply.

Unregistration (or disconnection)

When a Remote Service unregisters (or the connection between the Remote Service and the server is lost) then the Domain Topic (and thus all subdomain Topics) is automatically removed at the server (edge and relay).

Remote Control Publishers

Remote Control Publishers allow communication from Remote Services.

Enabling Remote Control within a Publisher is very simple. The Publisher creates a 'Remote Control Topic' that uses 'Remote Control Topic Data'. This allows a Remote Service to connect and register itself which causes a 'Domain Topic' that has Remote Service Topic Data to be created. The Remote Service may then create and manage further subdomain Topics subordinate to the Domain Topic.

Remote Control Topics

A Remote Control Topic is one to which registration requests from Remote Services will be routed. Such a Topic would be created with a name that is known to the Remote Service applications. Many different Remote Services can connect to a single Server via one Remote Control Topic.

In order to implement Remote Control at a Server all that is necessary is for a Publisher to add a Topic that has Remote Control Topic Data. Exactly how this should be done depends upon whether a two or three tier architecture is required.

Two Tier

In a two tier architecture there is only an edge Publisher and in order to enable Remote Control in this Publisher all that is necessary is to create a Topic that has Remote Control Topic Data as shown in the following example:-

```
public class RemoteEdgePublisher1 extends Publisher {

    public static final String CONTROL_TOPIC = "RemoteService";

    /**
     * @see Publisher#initialLoad()
     */
    @Override
    protected void initialLoad() throws APIException {
        RemoteControlTopicDataEdge topicData =
            TopicDataFactory.newRemoteControlData();

        addTopic(CONTROL_TOPIC,topicData);
    }
}
```

A Publisher could act as a Remote Control edge Publisher with no more code than shown above.

Remote Services connect to the Server hosting such a Publisher and register specifying the name of the Remote Control Topic (e.g. " `RemoteService` ").

The Remote Control Topic needs to exist before Remote Services can register. It will handle the creation of Remote Service Topics on behalf of Remote Services and will also ensure that all requests for subdomain Topics are routed

Copyright 2013 Push Technology

back to the correct Remote Service. If the connection to a Remote Service is lost then its Domain Topic (and thus all subdomain Topics) will automatically be removed.

Three Tier

In a three tier architecture there is an edge Publisher which interfaces with the Clients and connects to a relay Publisher (via a Publisher Client connection).

The edge Publisher must create a Topic with Remote Control Topic Data that specifies the location of the relay.

For example:-

```
public class RemoteEdgePublisher2 extends Publisher {

    public static final String CONTROL_TOPIC = "RemoteService";

    private static final String RELAY_HOST = "localhost";
    public static final int RELAY_PORT = 8081;

    /**
     * @see Publisher#initialLoad()
     */
    @Override
    protected void initialLoad() throws APIException {
        ServerDetails serverDetails =
            ConnectionFactory.createServerDetails(
                "dpt://"+RELAY_HOST+":"+RELAY_PORT);
        serverDetails.setInputBufferSize(256*1024);
        serverDetails.setOutputBufferSize(256*1024);

        RemoteControlTopicDataEdge topicData =
            TopicDataFactory.newRemoteControlData();

        RemoteControlRelayDetails relay =
            new RemoteControlRelayDetails(serverDetails);
        relay.setOutboundMessageQueueDefinition("relayQueue");
        topicData.setRelay(relay);

        addTopic(CONTROL_TOPIC,topicData);

    }
}
```

Note that the settings relating to buffer sizes and the queue definition to use for outbound messages are important to the performance of the relay connection.

At the relay tier there must be another Publisher that creates a Remote Control Topic with exactly the same name as at the edge but this time its Topic Data must be relay specific:-

```
public class RemoteRelayPublisher extends Publisher {
    /**
     * @see Publisher#initialLoad()
     */
    @Override
    protected void initialLoad() throws APIException {

        RemoteControlTopicDataRelay topicData =
            TopicDataFactory.newRemoteControlRelayData();

        addTopic(RemoteEdgePublisher2.CONTROL_TOPIC,topicData);
    }
```

Copyright 2013 Push Technology

```
}
```

At its simplest a three tier Remote Control Publisher architecture can be deployed with no more code than shown above.

The relay Publisher should be started first so that the edge can connect to it. The edge can then be started and will connect to the relay. If the relay is not available when the edge starts it employs a retry policy to keep trying to connect to the relay.

Remote Services will connect to the relay server in this case. Before Remote Services can register a valid edge connection must have been established.

The relay Topic Data handles the routing of all messages between Remote Services and edge. The relay Topic Data will create and maintain Remote Service Topics and subdomain Topics at the relay tier on behalf of Remote Services. These Topics are only used to mirror the edge Topic tree so that messages can be routed accordingly.

If the connection between a Remote Service and a relay Publisher is lost then the Remote Service Topic (and thus all subdomain Topics) is automatically removed and this in turn results in the removal of the same at the edge.

If the connection between the edge and the relay is lost then all Remote Service Topics in both tiers will be removed and Remote Services will be notified of the loss of the server connection.

Pre-loading Metadata

It is possible to pre-load metadata definitions that can be used by Remote Services when adding Topics. This is done using the `addMetadata` method.

Service Listeners

A listener for service related events can be declared using the `addListener` method. Such a listener would receive notifications on the following methods:-

Table 47: Adding Listeners

`serviceRegistered`	Called when a new Remote Service is registered.
`serviceUnregistered`	Called when a Remote Service is unregistered
`messageFromService`	Called when a message is received from a Remote Service (using its `sendToPublisher` method). A reference to the Remote Service Topic Data is passed which would allow a reply to be sent to the Remote Service if required.

In each case a reference to the Remote Service Topic Data is passed.

Remote Service (Domain) Topics

Remote Service Topics are created automatically by Remote Control Topic Data as a result of a Remote Service registering. These Topics act as the root of the service's domain and cannot be used for any other purpose.

There is no need for a Publisher to remove such a Topic as it is automatically removed when the connection with its Remote Service goes away.

Subdomain Topics

Subdomain Topics are those created subordinate to Service Topics. These are created by the Remote Service Topic Data as a result of a Remote Service adding Topics. Topics must never be added to Remote Service Topics in any other way.

Subdomain Topics are normal Topics other than that they are managed by a Remote Service. All requests from Clients for them (e.g. subscription, fetch, service etc) are automatically routed to the Remote Service for processing. The

Copyright 2013 Push Technology

Remote Service also publishes or communicates with Clients on these Topics. Publishers should not attempt to access subdomain Topics directly.

Publisher Considerations

Currently it is not possible to use Topic Aliasing with Remote Control Publishers because the faclity operates over the Event Publisher protocol which does not support aliasing. It is therefore important that Topic Aliasing is turned off in the Publisher configuration.

Configurable Remote Control Publishers

Two remote control publishers are included that provide a minimal, configurable remote control implementation. The `RemoteControlPublisher` is the edge publisher of the two and three tier architectures. The `RemoteControlRelayPublisher` is the relay publisher of the three tier architecture. Both can be configured through properties in the *`etc/Publishers.xml`* .

All the properties have default values that should allow the two tier architecture to run. The three tier architecture will require the edge Publisher to be configured with the details of the relay Publisher so that a connection between them can be established. Both the relay and the edge can be configured with a control topic name by the property `control.topic` , which defaults to RemoteControl. This must be the same for both the relay and the edge. The edge may also be configured to use a relay. To use a relay the property `relay` must be true, it defaults to false. With a relay enabled then the relay's host, port, input buffer size, outbound queue and retry interval should also be defined using the properties `relay.host` , `relay.port` , `relay.input.buffer.size` , `relay.outbound.queue` and `relay.retry.interval` .

Copyright 2013 Push Technology

Copyright 2013 Push Technology

Chapter 16

System Management

Topics:

- *General Management*
- *Classic Deployment*
- *Hot Deployment*
- *Using JMX*
- *Enabling collection of Statistics*
- *Diffusion Monitoring Console*
- *Basic integration with Splunk*

This section discusses how to manage a Diffusion™ Server and System as a whole.

General Management

Starting the Diffusion™ Server

The Diffusion™ Server is started using the "`diffusion.bat`" (or "`diffusion.sh`" on Unix platforms) script in the issued bin directory.

Configuration files are loaded by default from the `etc` directory. An optional properties directory can be specified as a parameter to the `diffusion` command. If a properties directory is specified then any properties files in the specified directory will be loaded in preference to those in the `etc` directory. The Server will log where it loads each property file from.

Classic Deployment

Installing Publishers into a stopped Diffusion™ instance.

The Publishers that will be started when Diffusion™ starts must be defined in the configuration file *`etc/Publishers.xml`* . You can see details of this in 'Defining Publishers' and 'Creating a Publisher Class'. Publishers that will not start with Diffusion™ can also be defined in the *`etc/Publishers.xml`* and these can be started later using JMX.

The Publishers must be present on the classpath of the Diffusion™ server. The recommended way to do this is to compile the Publishers source code with the diffusion.jar they will run with on the classpath. Then package the Publisher class files into a JAR file. This JAR file should then be deployed to the `ext/` directory of the Diffusion™ installation. The Diffusion™ server will search the `ext/` directory and load all the JAR files it finds.

Hot Deployment

Installing Publishers into a running Diffusion™ instance.

In addition to starting Publishers by defining them in *`etc/Publishers.xml`* , they may be installed into an already running Diffusion™ instance by a process known as hot deployment. Publishers may also be undeployed and redployed, providing they implement the `isStoppable` method, and it returns true. It is possible to also deploy dependant JAR files, configuration files and associated web pages for a Publisher. All artefacts required for deployment are packaged within a DAR file.

Deployment Methods

There are currently two ways to deploy a DAR file;

1. **File copy**

   Using this technique, a DAR may be copied to the deployment directory on the file system. By enabling auto-deployment in *`etc/Server.xml`*, Diffusion™ will periodically scan a directory for new or updated DAR files and deploy them. In the case of an updated DAR, the existing publisher will be undeployed (if possible) before being redeployed.

2. **HTTP**

   The DAR file may also be `POST`ed over HTTP if the deploy-service is enabled in *`etc/Publishers.xml`* . Command line tools such as curl can then be employed, for example:

   ```
   curl --data-binary @MyPublisher.dar http://localhost:8080/deploy
   ```

   Note that the HTTP connector must be configured to have an input buffer large enough to contain the entire DAR file.

   Undeployment

Copyright 2013 Push Technology

For Publishers deployed using the file copy method, the DAR file maybe deleted from the deployment directory and on the next scan, it will be undeployed. A DAR file can only be undeployed if all of the publishers it contains are stoppable. If a DAR file fails to be undeployed any future modifications to it will be ignored.

It is important that any files that the deployment process has extracted from the DAR are not deleted until the Publisher has been successfully undeployed. Publishers may also be undeployed via JMX by invoking the undeploy operation on associated MBean, for example

```
localhost/Server/com.pushtechnology.diffusion - Publisher - MyPublisher -
 undeploy()
```

Anatomy of a DAR file

What is in a DAR file?

A DAR file will contain at least three files:

- META-INF/MANIFEST.MF
- etc/Publishers.xml
- ext/MyPublisher.jar*

*Substitute with the name of your Publisher

An example folder structure within a DAR file may look like this:

The `MANIFEST.MF` file contains an attribute, `Diffusion-Version`, which specifies the minimum version number of Diffusion™ on which this Publisher will run. This prevents deployment of Publishers to Diffusion™ instances which may not support features of the Publisher, or have different API signatures.

```
Manifest-Version: 1.0
Diffusion-Version: 4.0.1
```

The `etc` directory contains files which would normally be found in Diffusion™'s `etc` directory, but will only contain information relating to the Publisher being deployed. Files that affect the operation of Diffusion™ and have no relationship to the Publisher are not loaded. Valid configuration files are:

- *`etc/Aliases.xml`* For when there are associated HTML files
- *`etc/Publishers.xml`* (mandatory)
- *`etc/SubscriptionValidationPolicy.xml`* If referenced in Publishers.xml

The `Publishers.xml` file follows the same pattern as in Diffusion™'s `etc` directory, for example:

```
<publishers>
        <publisher name="MyPublisher">
```

Copyright 2013 Push Technology

```
 <class>com.pushtechnology.diffusion.test.publisher.MyPublisher</class>
                <start>true</start>
                <enabled>true</enabled>
        </publisher>
</publishers>
```

You should place all Java code required by your Publisher in the '`ext`' folder. All files within this directory will be placed on the classpath for the Publisher, so this is where you would also add any required third-party JAR files or resources. The following `build.xml` for ant will take a Publisher developed as an Eclipse project, package and deploy to Diffusion™ via the filesystem method:

```
<project name="MyPublisher" default="deploy">
        <property name="publisher.name" value="MyPublisher" />
        <property name="jar.name" value="${publisher.name}.jar" />
        <property name="diffusion.dir" value="/home/diffusion/
Diffusion4.4.0" />
        <property name="dar.name" value="${publisher.name}.dar" />

        <target name="makejar">
                <jar jarfile="${jar.name}" includes="**/*.class"
 basedir="bin" />
        </target>

        <target name="makedeployable" depends="makejar">
                <delete dir="deploy" />
                <mkdir dir="deploy/${publisher.name}" />
                <copy todir="deploy/${publisher.name}">
                        <fileset dir=".">
                                <include name="data/**" />
                                <include name="etc/**" />
                                <include name="ext/**" />
                                <include name="html/**" />
                        </fileset>
                </copy>
                <copy todir="deploy/${publisher.name}/ext"
 file="${jar.name}" />
                <jar jarfile="${dar.name}" includes="${publisher.name}/**"
 basedir="deploy" manifest="META-INF/MANIFEST.MF" />
        </target>

        <target name="deploy" depends="makedeployable">
                <mkdir dir="${diffusion.dir}/deploy" />
                <copy todir="${diffusion.dir}/deploy" file="${dar.name}" />
        </target>
</project>
```

As an alternative to deployment by copying files, the following may be used if `curl` is installed and the deployment service has been enabled.

```
        <target name="deploy" depends="makedeployale">
        <mkdir dir="${diffusion.dir}/deploy" />
        <exec executable="curl">
                <arg value="-X" />
                <arg value="POST" />
                <arg value="http://localhost:8080" />
                <arg value="--data-binary" />
                <arg value="@${dar.name}" />
        </exec>
        </target>
```

Copyright 2013 Push Technology

Using JMX

Diffusion™ may be managed using JMX (Java Management Extensions).

Diffusion binds to the TCP ports configured in `etc/Management.xml` to listen for *JMX* clients such as JConsole and JVisualVM. The properties specify a host name and *RMI* registry port to which Diffusion will listen for JMXRMI connections. The connection port relates to the normally ephemoral port employed by JMXRMI, this is normally only useful when configuring firewalls.

An admin user may be configured as required. Read-only users may be configured that can view but not manage the system in any way.

Mbeans

Registered MBeans with JMX service

Diffusion™ registers MBeans for many of it's principal features with the JMX service. Annotations on each of the MBeans employed are used to produce the following pages in this manual as well as feeding JMX clients with descriptive information. MBeans, attributes and operations have descriptions; operation arguments have names; operations also have JMX impact information. For example...

Figure 29: The Server MBean 'stopController' operation showing in JConsole

ClientConnectionStatisticsMXBean

Monitoring interface to the client connection statistics MBean

Attributes

Name	Type	Read/Write	Description
clientOutputFrequency	long	read-write	statistics output frequency in milliseconds
clientResetFrequency	long	read-write	the frequency with which the counters are reset
concurrentClientCount	int	read	The current client count
connectionCounts	Map	read	The current client count, broken down by client type

Copyright 2013 Push Technology

Name	Type	Read/Write	Description
maximumConcurrentClientCount	int	read	The maximum number of concurrent clients
maximumDailyClientCount	int	read	The count of connections recieved in a day

StatisticsServiceControllerMBean

Interface for StatisticsService controller

Attributes

Name	Type	Read/Write	Description
clientStatisticsEnabled	boolean	read-write	See if aggregate statistics for Clients are enabled
enabled	boolean	read-write	See if the server-wide Statistics service is enabled
publisherStatisticsEnabled	boolean	read-write	See if aggregate statistics for Publishers are enabled
serverStatisticsEnabled	boolean	read-write	See if aggregate statistics for Event Publishers are enabled
topicStatisticsEnabled	boolean	read-write	See if aggregate statistics for Topics are enabled

StatisticsManagerMBean

Monitoring interface to the collection of server statistics

Attributes

Name	Type	Read/Write	Description
statisticsEnabled	boolean	read-write	Global statistics switch

LogFileMXBean

Management interface for Log Definition

Attributes

Name	Type	Read/Write	Description
description	LogDescription	read	the LogDescription for this log
filename	String	read	The fully qualified filename of this log
logLevel	String	read-write	The current level as a string

VirtualHostMBean

HTTP Virtual Host management interface

Attributes

Name	Type	Read/Write	Description
cacheSizeBytes	int	read	the cache size in bytes
cacheSizeEntries	int	read	the number of entries in the cache
aliasFile	String	read	the alias file name

Copyright 2013 Push Technology

Name	Type	Read/Write	Description
aliasProcessor	HTTPAliasProcessor	read	the alias processor object
cache	HTTPCache	read	the HTTP cache
compressionThreshold	int	read	the compression threshold
debug	boolean	read	true if debug is set
documentRoot	String	read	the document root directory
errorPage	String	read	the error-page file name
fileServiceName	String	read	the file service name
homePage	String	read	the home-page file name
host	String	read	the host name
minify	boolean	read	true if the minify property is set
name	String	read	the virtual host name
numberOfRequests	int	read	number of requests actioned since service was started
static	boolean	read	true if static
webServerName	String	read	the web server name

Operations

Name	Return Type	Arguments	Impact	Description
startService	void	0	ACTION	Restart a previously stopped virtual host

Name	Return Type	Arguments	Impact	Description
stopService	void	0	ACTION	Stops the virtual host from processing requests

Name	Return Type	Arguments	Impact	Description
clearCache	void	0	ACTION	clear the cache of all entries

AuthorisationManagerInstanceMBean

Management interface to the optional AuthorisationManager

Attributes

Name	Type	Read/Write	Description
connections	int	read	Number of connections authorised
connectionsDenied	int	read	Number of connections denied authorisation
fetches	int	read	Number of fetches authorised
fetchesDenied	int	read	Number of fetches denied authorisation

Copyright 2013 Push Technology

Name	Type	Read/Write	Description
handlerClassName	String	read	Class name of any configured AuthorisationHandler
hasHandler	boolean	read	true if this server has an AuthorisationHandler configured
subscriptions	int	read	Number of subscriptions authorised
subscriptionsDenied	int	read	Number of subscriptions denied authorisation
writes	int	read	Number of writes authorised
writesDenied	int	read	Number of writes denied authorisation

JMXAdapterMXBean

Management interface to the adapter that reflects MBean attributes and notifications as Diffusion Topics

Attributes

Name	Type	Read/Write	Description
updateFrequency	long	read	MBean attribute poll frequency, in milliseconds

StatisticsHandlerMBean

Interface for a specific StatisticsHandler

Attributes

Name	Type	Read/Write	Description
instanceStatisticsEnabled	boolean	read-write	Instance-specific statistics for this Handler

TopicImplMBean

Deprecated monitoring interface to a Diffusion Topic

Attributes

Name	Type	Read/Write	Description
currentNumberOfSubscribers	int	read	Number of clients currently subscribing
outboundAverageMessageSize	long	read	Average size of sent messages
outboundNumberOfMessages	long	read	Number of messages sent via this Topic
publisherName	String	read	Name of the Publisher to which this Topic belongs
reference	String	read	Associated descriptive string reference
totalNumberOfSubscribers	long	read	Number of clients that have ever subscribed

ConnectorManageableMBean

Management interface to a Connector

Copyright 2013 Push Technology

Attributes

Name	Type	Read/Write	Description
keepAliveQueueMaximumDepth	int	read-write	the maximum queue depth used for clients in the keep alive state
keepAliveTime	long	read-write	the time in milliseconds that a unexpectedly disconnected client should be kept alive before closing
numberOfAcceptors	int	read-write	the number of Acceptors
queueDefinition	String	read-write	the queue definition
totalNumberOfConnections	long	read	number of connections accepted since the Connector was started
uptime	String	read	a pretty printed string of the time this connector has been running, or 0 if not
uptimeMillis	long	read	the milliseconds this connector has been running, or 0 if not

Operations

Name	Return Type	Arguments	Impact	Description
remove	void	0	ACTION	Remove the Connector. It will not be possible to restart the Connector again (until system restart).

Name	Return Type	Arguments	Impact	Description
start	void	0	ACTION	Start the connector.

Name	Return Type	Arguments	Impact	Description
stop	void	0	ACTION	Stop the connector. Allows it to be restarted.

DiffusionControllerMBean

Diffusion Controller

Attributes

Name	Type	Read/Write	Description
buildDate	String	read	The build date and time
freeMemory	long	read	The amount of free memory in the Java Virtual Machine
licenseExpiryDate	Date	read	The license expiry date
maxMemory	long	read	The total amount of memory in the Java virtual machine
numberOfTopics	long	read	The number of topics on this server
release	String	read	The Diffusion release string, e.g. 4.5.2_01

Copyright 2013 Push Technology

Name	Type	Read/Write	Description
startDate	Date	read	Time at which this Diffusion server began
startDateMillis	long	read	The time at which this Diffusion server was started, as milliseconds since the epoch
timeZone	String	read	name of the time zone to which this Diffusion server belongs
totalMemory	long	read	The total amount of memory in the Java virtual machine
uptime	String	read	The time that this controller has been running, e.g. "3 hours 4 minutes 23 seconds"
uptimeMillis	long	read	The time this controller has be running, expressed in milliseconds
usedPhysicalMemorySize	long	read	Free physical memory, expressed in bytes
usedSwapSpaceSize	long	read	Used swap space, expressed in bytes
userDirectory	String	read	The directory in which the Diffusion server was begun
userName	String	read	Name of the user to which the Diffusion server belongs

Operations

Name	Return Type	Arguments	Impact	Description
stopController	void	0	ACTION	Stop this Diffusion controller

Name	Return Type	Arguments	Impact	Description
stopController	void	2	ACTION	Stop this Diffusion controller, and record the reason and adminName

Argument name	Type	Description
reason	String	Reason this server is stopping
adminName	String	Name of the administrator

EventDrivenMultiplexerMBean

Management interface to the event driven multiplexer

Attributes

Name	Type	Read/Write	Description
latencyWarningTime	long	read	The latency threshold of this multiplexer, after which notifications shall be sent
name	String	read	The Multiplexer name
numberOfClients	int	read	The current number of clients assigned to multiplexer

Copyright 2013 Push Technology

MultiplexerManagerMBean

Management interface to the multiplexer manager

Attributes

Name	Type	Read/Write	Description
numberOfMultiplexers	int	read	the number of multiplexers

BaseThreadPoolMBean

Management interface to basic thread pool

Attributes

Name	Type	Read/Write	Description
activeCount	int	read	The number of active threads
coreSize	int	read-write	The number of threads to maintain
keepAlive	long	read-write	keep alive time in milliseconds
largestSize	int	read	the largest number of threads that have simultaneously been in the pool
maximumSize	int	read-write	Maximum Pool Size
queueLowerThreshold	int	read	The lower queue size at which the notification handler will be notified
queueMaximumSize	int	read	The maximum queue size that the task queue can reach
queueSize	int	read	The size of the embedded task queue
queueUpperThreshold	int	read	The upper queue size at which the notification handler will be notified
size	int	read	The current number of threads in the pool
taskCount	long	read	The approximate total number of tasks that have ever been scheduled for execution

EventPublisherMBean

Management inferface for an Event Publisher

Attributes

Name	Type	Read/Write	Description
totalNumberOfMessages	Long	read	the total number of messages processed
upTimeMillis	long	read	the time in milliseconds that the service has been running
uptime	String	read	the pretty-printed time that the service has been running

Copyright 2013 Push Technology

Operations

Name	Return Type	Arguments	Impact	Description
close	void	0	ACTION	Close the connection to this Event Publisher

PublisherControllerMBean

Management interface for a publisher

Attributes

Name	Type	Read/Write	Description
inboundClientAverageMessageSize	long	read	The average size of a message received from clients
inboundClientNumberOfMessages	long	read	The count of messages received from clients
inboundEventPublisherAverageMessageSize	long	read	The average size of a message received from Event Publishers
inboundEventPublisherNumberOfMessages	long	read	The count of messages received from Event Publishers
logLevel	String	read-write	The log level for this publisher
numberOfTopics	int	read	The count of topics associated with this Publisher
outboundAverageMessageSize	long	read	The average size of a message sent to clients
outboundNumberOfMessages	long	read	The count of messages sent to clients
started	boolean	read	true if the publisher is started

Operations

Name	Return Type	Arguments	Impact	Description
removePublisher	void	0	ACTION	Permanently remove the publisher

Name	Return Type	Arguments	Impact	Description
startPublisher	void	0	ACTION	Start this publisher

Name	Return Type	Arguments	Impact	Description
stopPublisher	void	0	ACTION	Stop this publisher

Name	Return Type	Arguments	Impact	Description
undeploy	void	0	ACTION	Un-deploy this publisher

JMX Adapter

Reflect JMX MBeans their properties and notifications as topics

The JMX Adapter is packaged within the Diffusion™ publisher, and exists to optionally reflect the state of the JMX MBeans (both built-in, Diffusion™ and 3rd party in origin) as topics. There are a considerable number of statistics made available as MBean properties (e.g. CPU load, OS version, number of file-descriptors, threads, etc.) and making

Copyright 2013 Push Technology

them also available to Diffusion™ clients makes possible the implementation of system monitoring solutions to the web, and all other Diffusion™ platforms.

Configuration/Enablement

The JMXAdapter is part of the Diffusion™ Publisher, and will only execute if the Diffusion™ publisher is running and is configured to do so with the following properties within the Diffusion™ publisher's configuration.

```
<!-- If the JMXAdapter is enabled, then JMX MBeans will be exposed as topics
 -->
<property name="JMXAdapter.enabled">true</property>
<!-- The refresh frequency in ms -->
<property name="JMXAdapter.refreshFrequency">3000</property>
```

When the JMX Adapter starts it listens for the registration and unregistration of MBeans and reflects any changes into the topic tree. This covers all MBeans: built-in, Diffusion™ and any 3rd party Beans. Topics that reflect MBean attributes can be subscribed to and it is the role of the JMX adapater to polls all subscribed attributes and update their matching topics if required. The poll frequency is governed by the publisher property 'JMXAdapter.refreshFrequency' and defaults to 5 seconds.

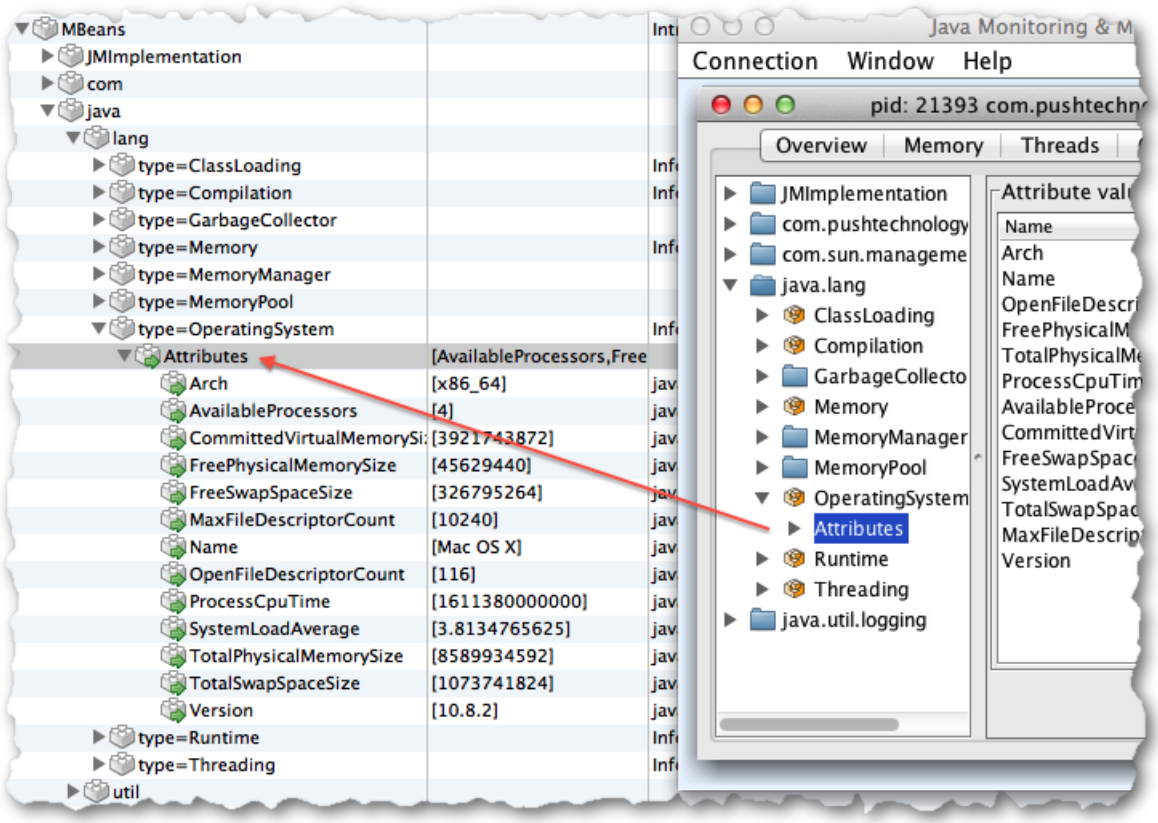


Figure 30: Reflecting MBeans as topics

MBean notifications are also available as topics. Whenever a notification is thrown and the matching topic is subscribed a message holding a number of key attributes is published

Table 48: Notifications as Topics

Record starting ...	Holding
message	javax.management.Notification.getMessage()

Copyright 2013 Push Technology

Record starting ...	Holding
sequenceNumber	javax.management.Notification,getSequenceNumber()
timeStamp	javax.management.Notification.getTimeStamp()
userData	javax.management.Notification.getUserData() if present
source	javax.management.Notification.getSource()

The JMX Adapter is itself an MBean with obect-name com.pushtechnology.diffusion:name=JMXAdapter, which exposes the polling freqency in milliseconds as attribute 'UpdateFrequency'. A value less than or equal to zero prevents polling.

MXBeans vs. Simple MBeans

The JMX adpater caters for both MXBeans and simpler MBeans. All MBean attributes are serialised as strings when converted to topics, this may be impractical if a solution needs to return an Object or an array of Objects. MXBean attributes with ArrayType, TabularType and CompositeType types are treated differently.

- CompositeType Fields within the composite attribute are mapped to discrete topics.

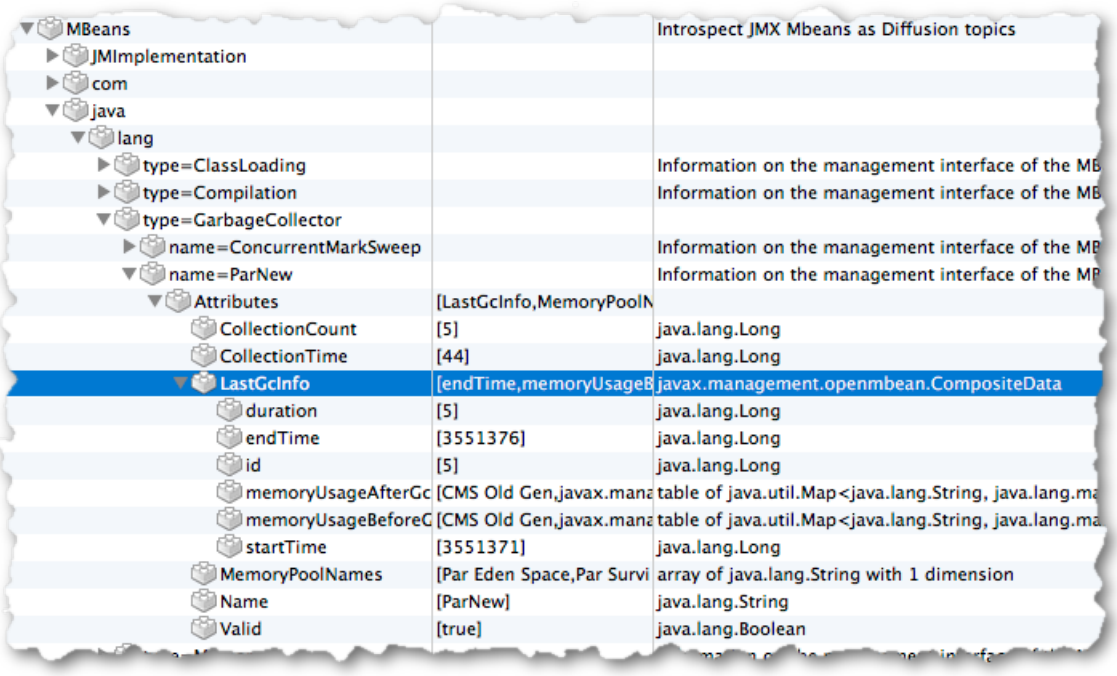


Figure 31: Showing a composite attribute as a topic nest

- TabularType Each entry in the attribute is rendered as a record, where the first field is the key-name and the second field the value rendered as a string. Keys are shown in sorted order. TabularType attributes holding non SimpleType attributes are not supported (e.g a TabularType attribute holding Composite or ArrayType values)
- ArrayType One dimensional arrays are presented as a single record with many values. Two dimensional arrays are converted to many records. ArrayType attributes holding non SimpleType attributes are not supported (e.g an ArrayType attribute holding Composite or ArrayType values)

Copyright 2013 Push Technology

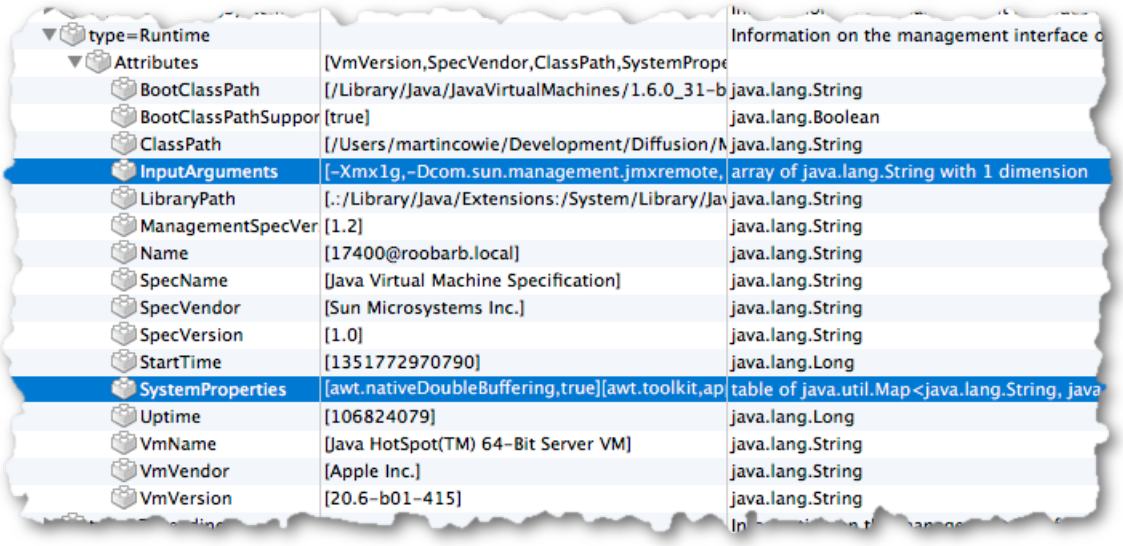


Figure 32: Topics reflecting an ArrayType and TabularType MXBean attributes

Using Visual VM

VisualVM is a good tool for seeing what is happening inside the Diffusion™ instance. This is sometimes installed with JDK's but can be down loaded from https://visualvm.dev.java.net/

This is sometimes installed with JDK's but can be down loaded from https://visualvm.dev.java.net/

Running Visual VM locally

Starting the instance of Visual VM locally will present a list of currently running Java processes.

Copyright 2013 Push Technology

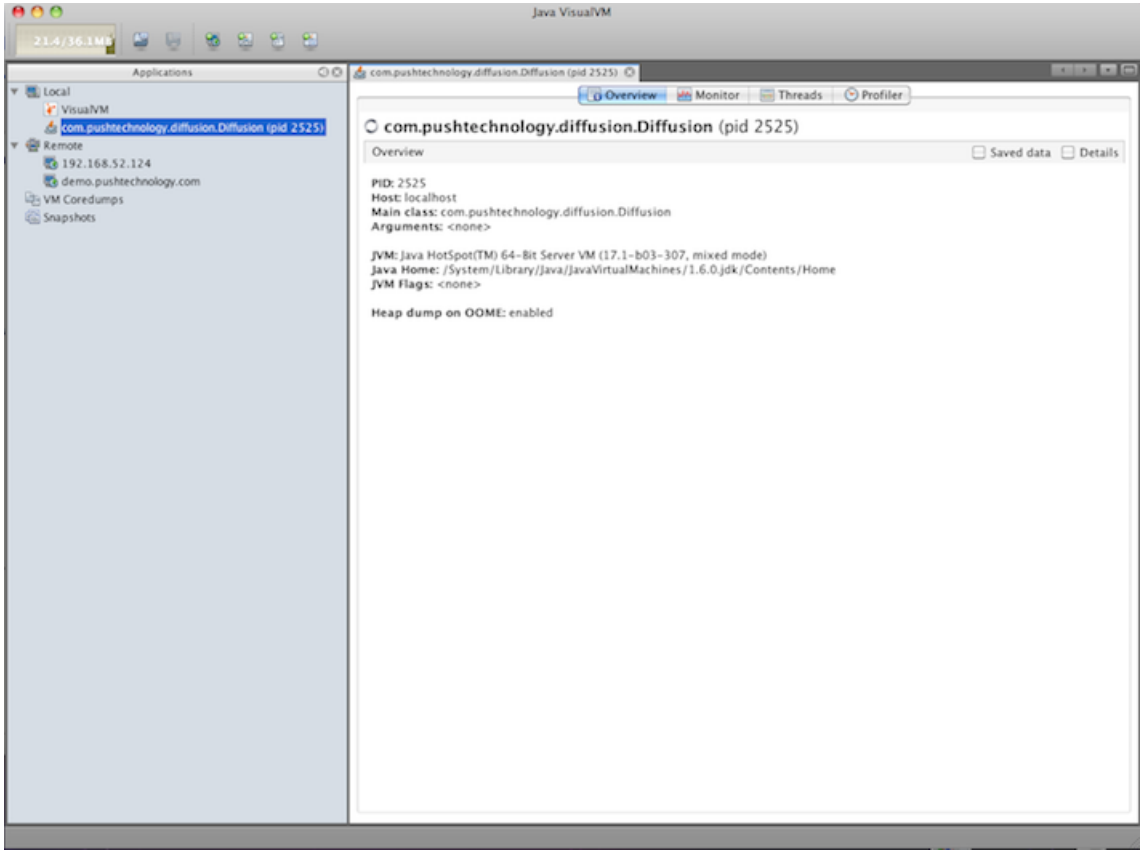


Figure 33: Java VisualVM

Its is then possible to select multiple views (tabs) detailing the JVM itself.

Monitor

Copyright 2013 Push Technology

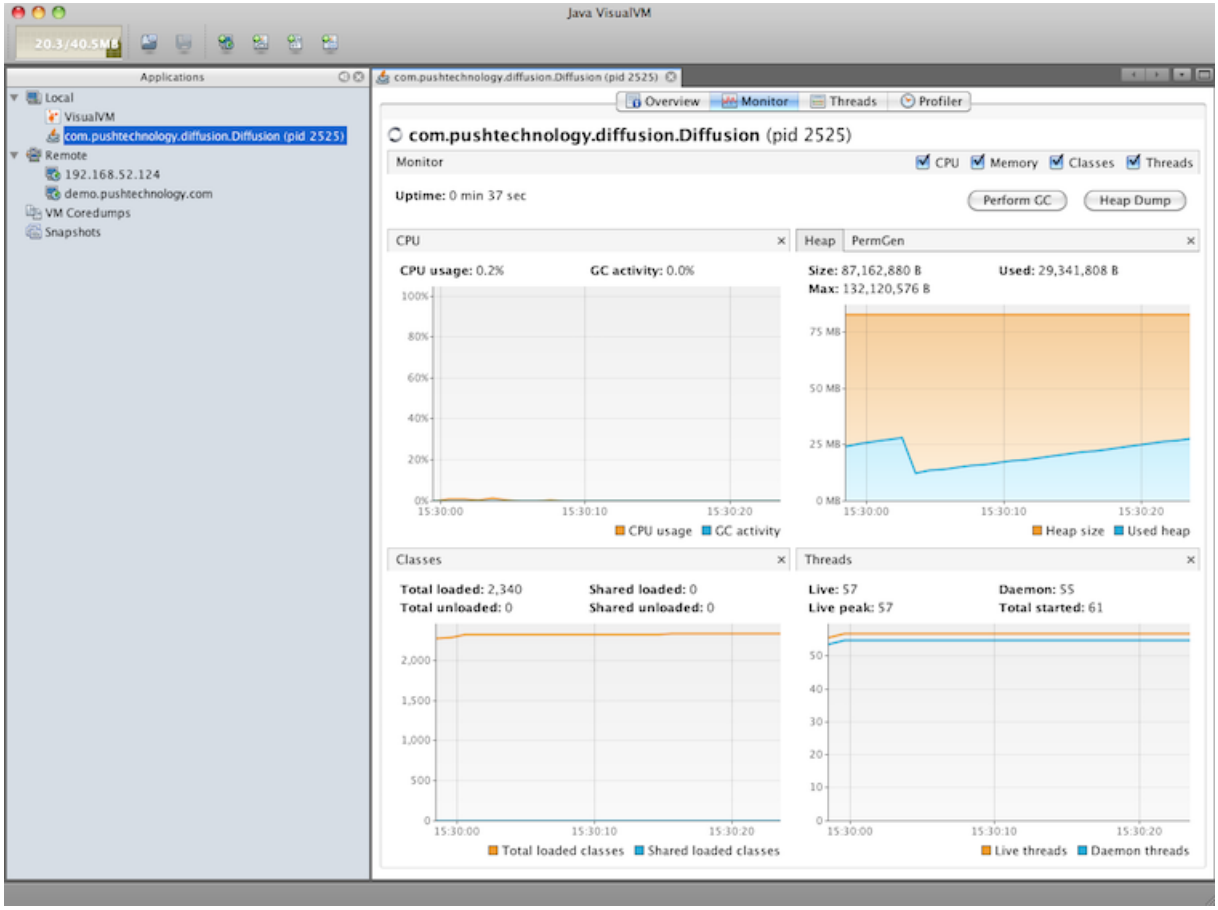


Figure 34: Java VisualVM monitor

Threads

Copyright 2013 Push Technology

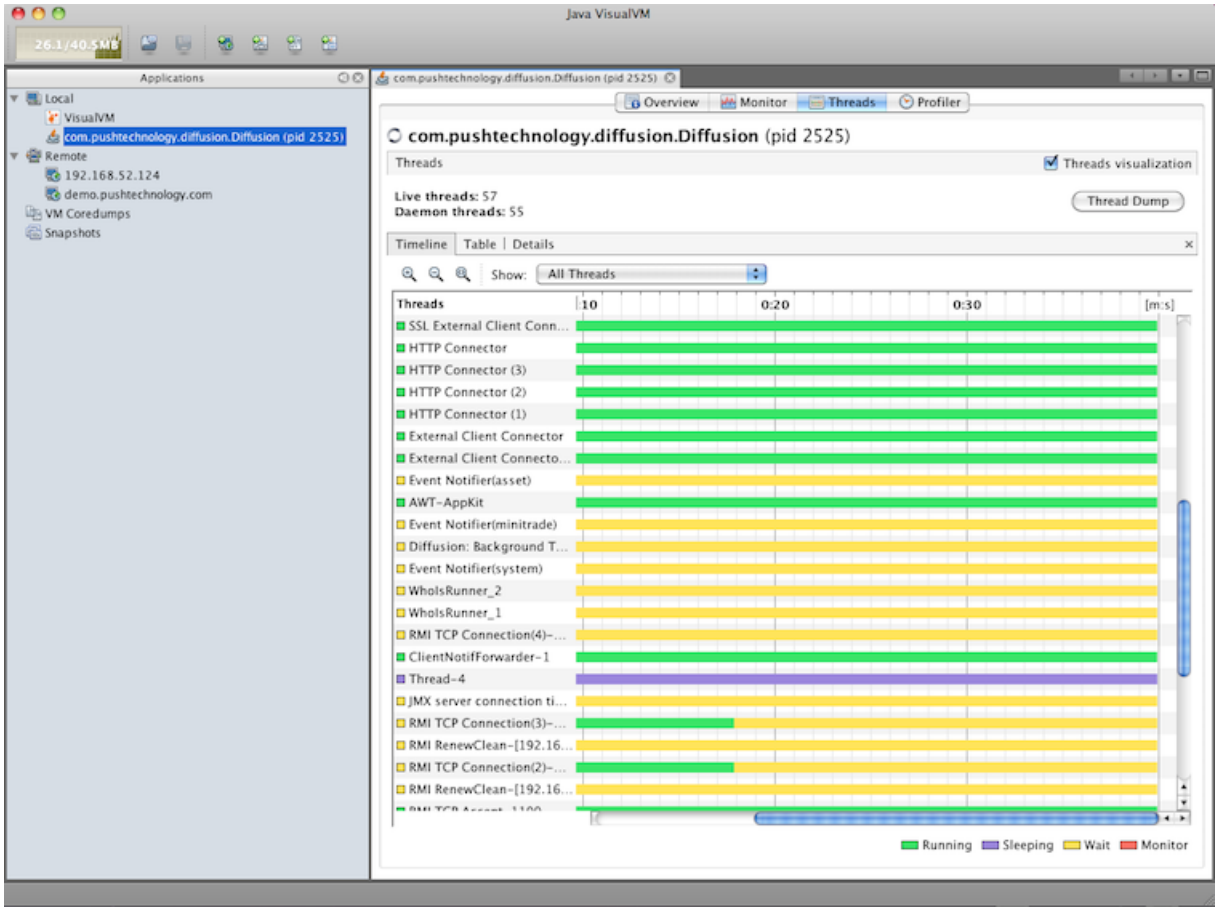


Figure 35: Java VisualVM threads

Profile

Copyright 2013 Push Technology

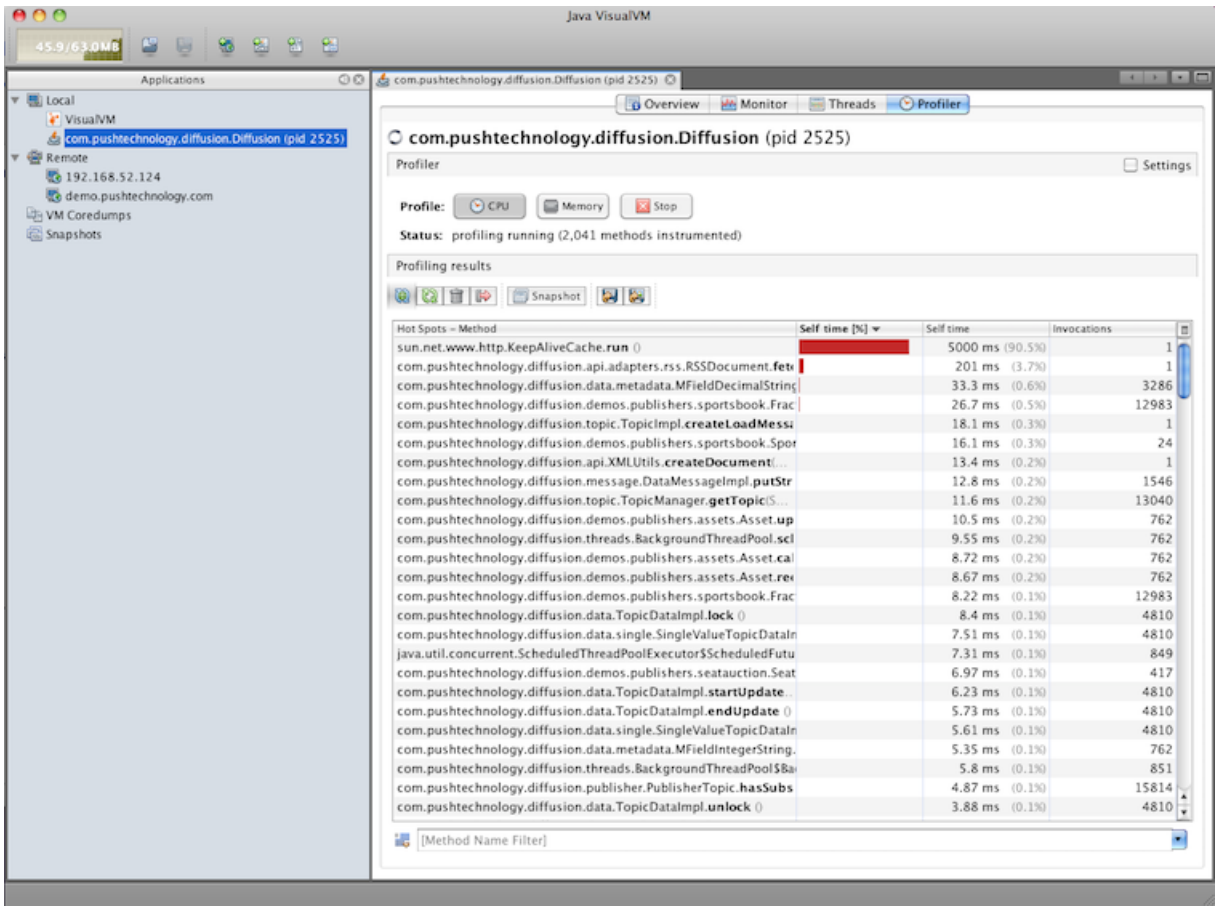


Figure 36: Java VisualVM profile

Running Visual VM remotely

To run Visual VM remotely, you will be required to run a process called Jstatd on the server. It may also be required to run the Jstatd process as the same user that owns the Diffusion™ process. To aid in this there is a jstatd(.sh/bat) shipped in the tools directory.

If you do not have the tools folder on the remote machine then you will need to start jstatd manually. To do this there needs to be a policy file in place. The policy file should contain the following..

```
grant codebase "file:${java.home}/../lib/tools.jar" {
   permission java.security.AllPermission;
};
```

Then you will need to add the following to the jstatd command.

```
 J-Djava.security.policy=jstatd.policy
```

When attaching to a remote instance it may be necessary to alter the Diffusion™ startup parameters to something like the following :

```
-Dcom.sun.management.jmxremote.port=3333
-Dcom.sun.management.jmxremote.host={host}
-Djava.rmi.server.hostname={host}
-Dcom.sun.management.jmxremote.authenticate=false
-Dcom.sun.management.jmxremote.ssl=false -Dcom.sun.management.jmxremote
```

Copyright 2013 Push Technology

Note here that you can attach VisualVm to port 3333 and the normal JMX connection for Diffusion™ will be 1099 (or whatever is set in *etc/Management.xml*)

As with JConsole visual VM can fail silently and not gave any reasons for not connecting. If you do have difficulty in connecting then adding this extra argument both client and server can help diagnose issues.

```
-Djava.rmi.server.logCalls=true
```

Using JConsole

Diffusion™ may be managed via the JMX system management console JConsole.

The properties specify a host name an RMI port and a connection port on which will listen for JMX connections.

An admin user may be configured as required. Read-only users may be configured that can view but not manage the system in any way.

Connecting to the JMX service

Diffusion™ may be managed via a JMX system management console such as JConsole.

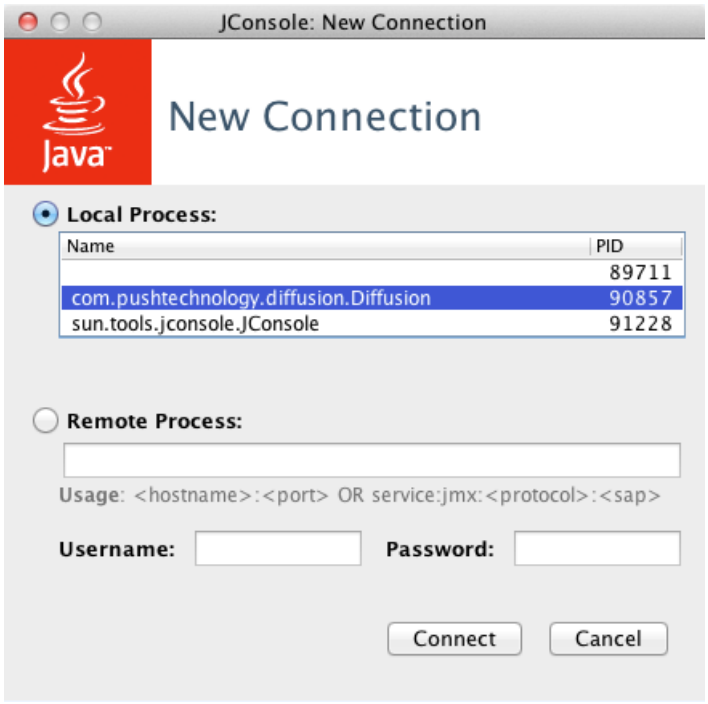


Figure 37: JConsole

Once connected to JMX, several aspects of the system are available to monitor and tune.

Copyright 2013 Push Technology

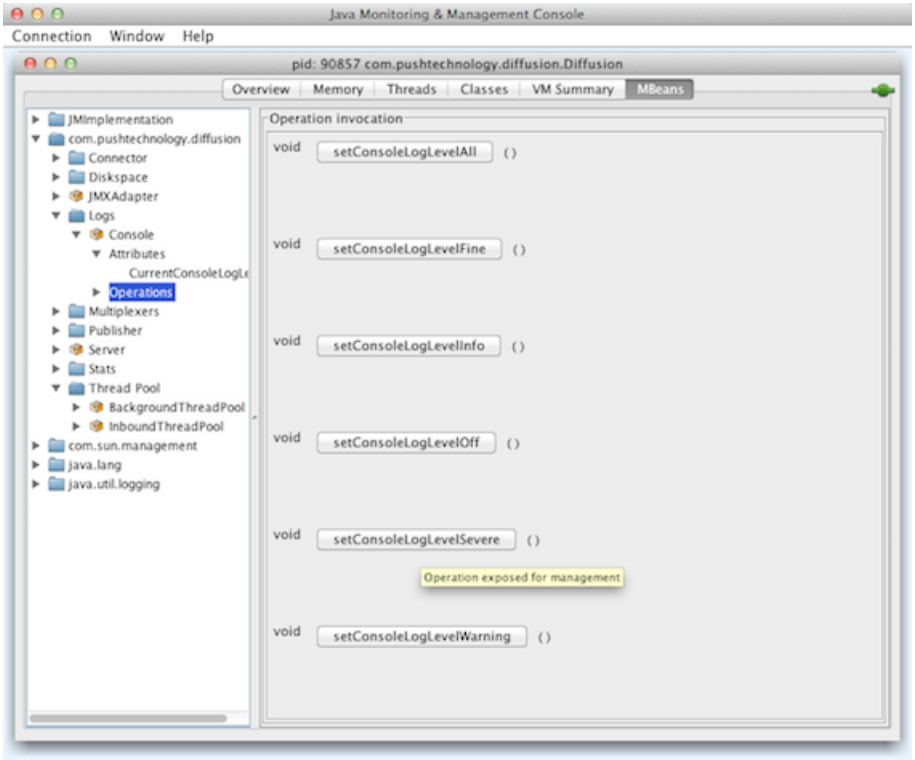


Figure 38: Tuning and monitoring

This enables many views (tabs) on what is going on inside the JVM.

JConsole can be quite problematic on remote connections, and it will fail silently. There are additional command line options that you can specify get get more logging on the connection. Adding `-J-Djava.util.logging.config.file=logging.properties` to the jconsole command line. The logging.properties file will need to be created with the following entries:

```
handlers= java.util.logging.ConsoleHandler.level=INFO

java.util.logging.FileHandler.pattern = %h/java%u.log
java.util.logging.FileHandler.limit = 50000
java.util.logging.FileHandler.count = 1
java.util.logging.FileHandler.formatter = java.util.logging.XMLFormatter

java.util.logging.ConsoleHandler.level = FINEST
java.util.logging.ConsoleHandler.formatter =
 java.util.logging.SimpleFormatter

// Use FINER or FINEST for javax.management.remote.level - FINEST is
// very verbose...
//
javax.management.level=FINEST
javax.management.remote.level=FINER
```

Often the remote end (Diffusion™ server in this case) gets confused about what its remote sever address is and you may have to add `-Djava.rmi.server.hostname={host or IP address}` to the Diffusion™ starting arguments.

Connecting to Diffusion on a remote server

Diffusion makes provision for users wishing to monitor and manage a remote Diffusion server with JConsole. The default configuration in *`etc/Management.xml`* establishes a listener on port 1099 for all network interfaces.

Copyright 2013 Push Technology

Consequently jconsole users input the host and port into JConsole's "Remote Process" control. The username and password in question are also held in *etc/Management.xml*

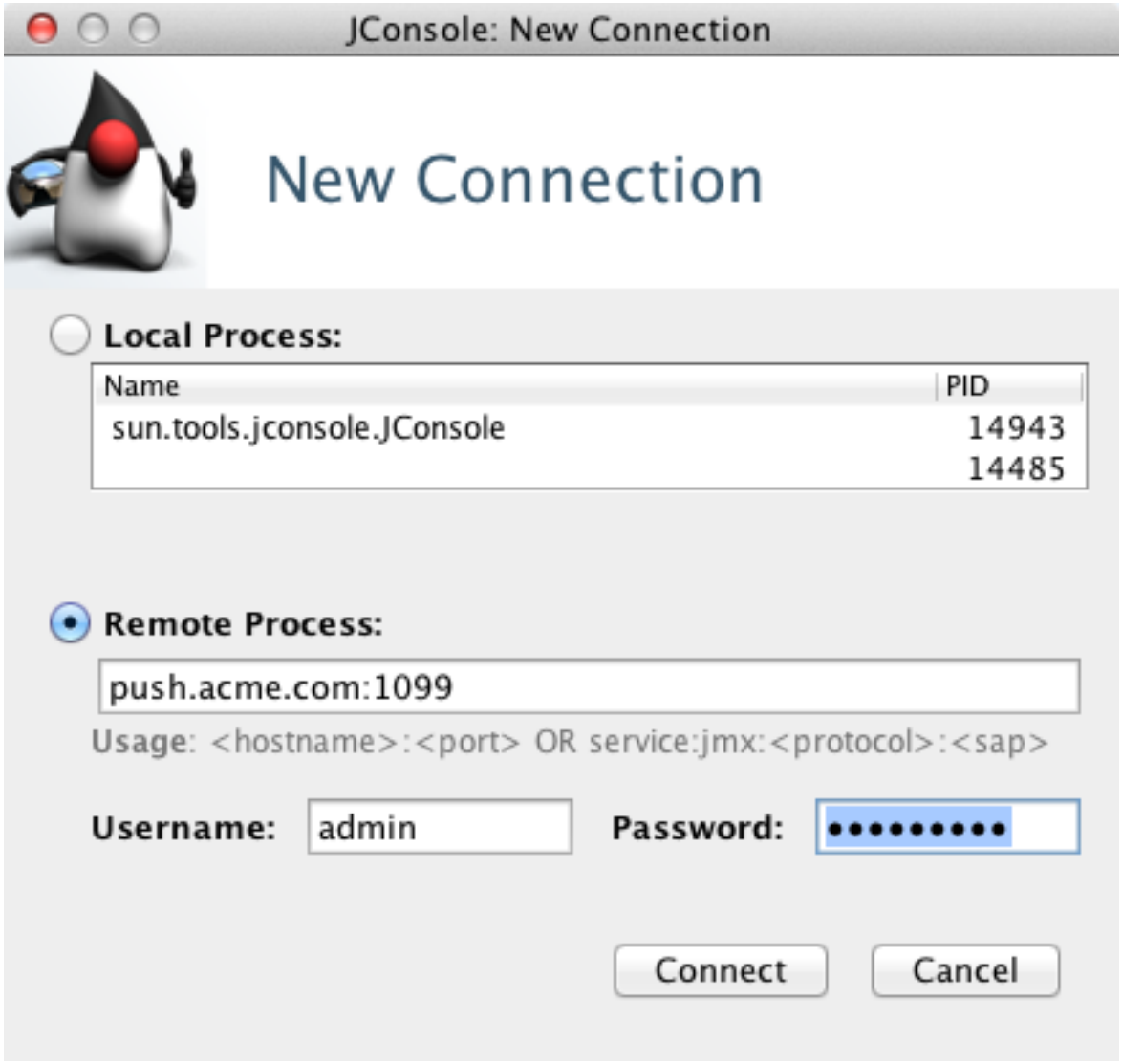


Figure 39: Monitoring a remote Diffusion server with JConsole

JConsole and Firewalls

The default configuration in *etc/Management.xml* makes the two port numbers predictably 1099 and 1100. You may find however that the if your firewall solution also employs *Network Address Translation*, (NAT) you may still be unable to connect to Diffusion even when TCP ports 1099 and 1100 are left open. This is due to a well understood weakness in the default protocol used by JConsole: the server redirects the client's initial connection to a second host and port, problematically it employs the IP address of the server's NIC, rather than the IP address the client connected to the first port on.

The more popular and more secure solution is to establish an *SSH* connection to the firewalled Diffusion server, tunnelling ports 1099 and 1100 through SSH, then using JConsole to connect to the local ends of the tunnelled ports.

Enabling collection of Statistics

How to enable Statistics

Copyright 2013 Push Technology

By default statistic collection is turned off for performance reasons, to enable statistic collection, set the statistics root attribute `enabled`in Log.xml to true.

Statistics

Collecting Statistics

Diffusion™ Servers provide statistics which are available via the JMX MBeans or via topics under the topic `Diffusion/Metrics`. If statistics is enabled via `etc/Statistics.xml` users can take measurements including the average message size, number of messages per topic per second, etc.

The Statistics Configuration has several distinct elements, that allow granular control over what is enabled on server start.

- `<statistics>`The top-level element; setting the 'enabled' property to false will disable all Statistics for the server
- `<topics>` `<clients>` `<publishers>` `<server>` Enabling these elements will provide aggregate metrics for each given class. Each element also contains a `<monitor-instances>` element that dictates whether specific instances of the parent class are monitored. Instance metrics require that the parent element is enabled.

The collection of Metrics is configured seperately from the distribution thereof. Within the Statistics.xml configuration file, there is also a `<reporters>` element which contains definitions of available 'reporters', which will expose metrics over different mediums.

Some reporters allow certain properties to be passed to them. For instance, the Topics reporter allows the use of `<property name="interval">` to provide the desired update frequency in seconds. Details of valid properies is documented within the `etc/Statistics.xml` file itself.

Programmatic configuration

It is programatically possible to query and configure the recording and calculation of statistics for the classes:

- `com.pushtechnology.diffusion.api.publisher.Client`
- `com.pushtechnology.diffusion.api.topic.Topic`
- `com.pushtechnology.diffusion.api.publisher.Publisher`

Developers are able to query the state of each class, to discover whether it is recording statistics using method `isStatisticsEnabled()`, stop recording with `stopStatistics()` and start recording using `startStatistics()`. The relevant Javadocs hold more detail on the subject.

It has been observed that in a system with significant numbers of clients enabling the client instance statistics can result in a performance impact of up to 20% reduction in maximum throughput achieved, which can inhibit the system from supporting further connections.

If your system has more than 20k clients (at any given time) per Diffusion instance we recommend the client instance statistics be turned off.

Similarly if your system supports very large number of topics turning on the topic instance statistics may result in a performance hit.

Diffusion Monitoring Console

A web console for monitoring your Diffusion solution

About

The Diffusion Monitoring Console is an optional publisher, provided as `console.dar`. It is deployed by default, and can be *undeployed* in the same manner as any DAR file. It exists to give operational staff using a web browser accessible visibility over the operations of a Diffusion solution

Copyright 2013 Push Technology

Users wishing to *manage* a Diffusion server, that is make changes to it, need still to use JMX tools such as JConsole unless it is limited to stopping the Diffusion server, and stopping and restarting a publisher.

Dependencies

The console depends on the Diffusion publisher to mirror JMX MBeans as topics. The console also makes use of the statistics controlled by `etc/Statistics.xml`

The live graphing feature mandates a web browser that supports Scalar Vector Graphics (SVG). Most modern web browser implement the features required by the Console however Internet Explorer v9 is the recommended minimum for Microsoft users.

There are also two configuration settings within the **Diffusion Publisher** config within `etc/Publishers.xml`. These are:

- `console.control.server` - Enable the ability to stop the Diffusion server through the console.
- `console.control.publishers` - Enable the ability to stop a particular Publisher through the console.

Both of these options are disabled by default.

Features: Dashboard

Dashboard Panels

By default the Console is deployed as part of Diffusion. It is available in a fresh installation at *http://localhost:8080/console/*.

Default layout

Started for the first time the console consists of six panels, each focussing on a key feature of the server

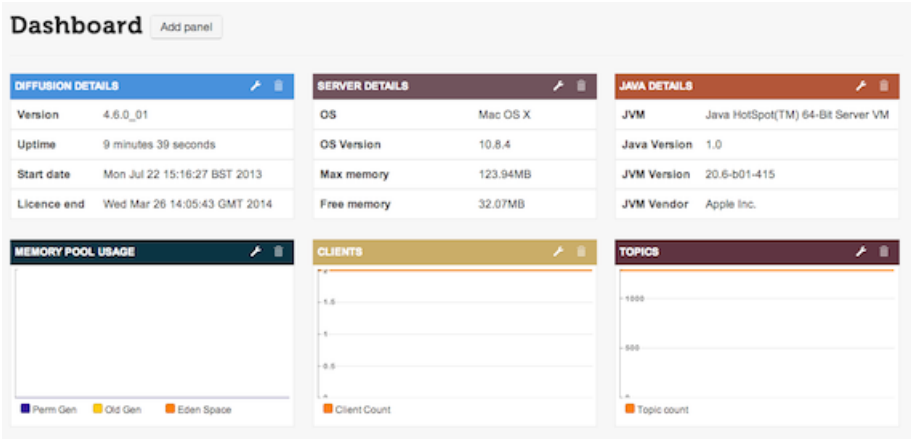


Figure 40: The default console layout

- **Diffusion Details**: the server version; the server ‘up time’, the server start date and time and the time and which the current license expires.
- **Server Details**: the name and version of the underlying operating system; the total memory available (physical and virtual) and the amount of free memory.
- **Java Details**: the name, vendor and version of the Java Virtual Machine.

Instead of tabular data the second row show live line graphs.

- **Memory Pool Usage**: the values over time of the memory used by the JavaVM process.
- **Clients**: the value over time of the number of Diffusion clients connected.
- **Topics**: the value over time of the number of topics on your Diffusion server.

Publishers Table

Copyright 2013 Push Technology

At the bottom of the Dashboard is the Publishers table. At a glance this shows the installed publishers and their vital statistics: the number of topics created, client connected, messages sent, bytes sent and finally publisher status

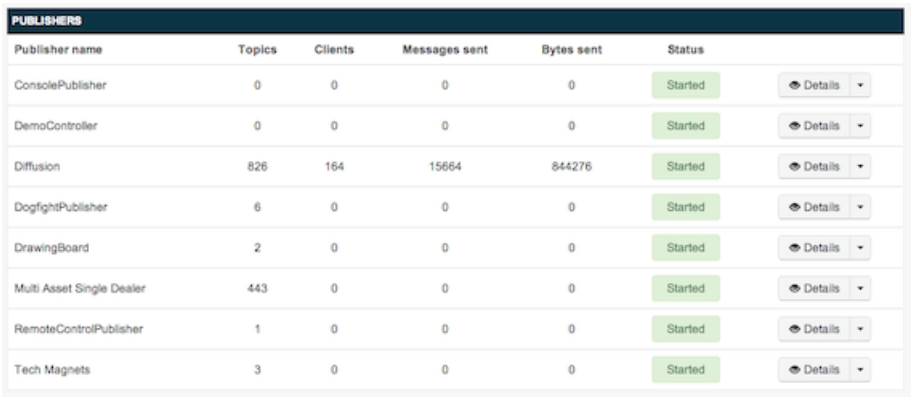

PUBLISHERS
Publisher name
Topics
Clients
Messages sent
Bytes sent
Status
ConsolePublisher 0 0 0 0 Started Details
DemoController 0 0 0 0 Started Details
Diffusion 826 164 15664 844276 Started Details
DogfightPublisher 6 0 0 0 Started Details
DrawingBoard 2 0 0 0 Started Details
Multi Asset Single Dealer 443 0 0 0 Started Details
RemoteControlPublisher 1 0 0 0 Started Details
Tech Magnets 3 0 0 0 Started Details

Figure 41: The table of Publishers

Using the pull-down menu on the "Details" button publishers can be stopped and restarted. The "Details" button itself reveals the publisher statistics: clients, topics, average messages per second and average bytes per second.

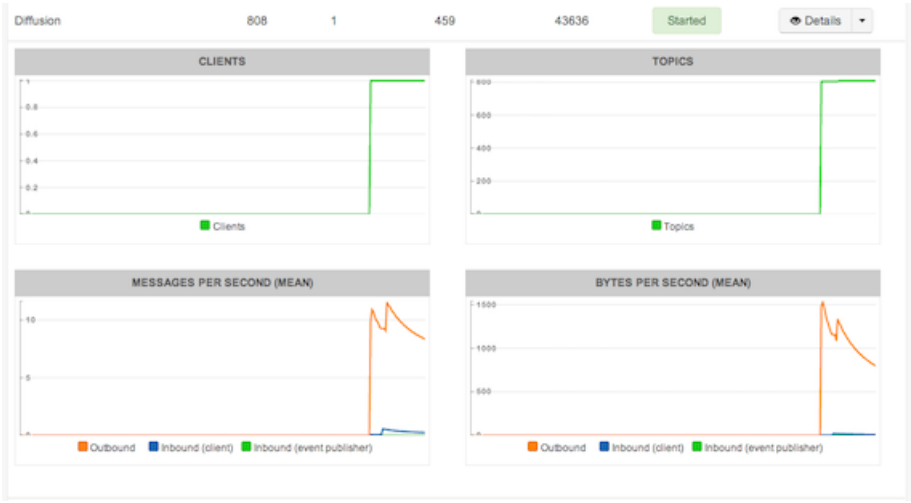


Figure 42: Publisher statistics graphs

Features: Topics Tab

The Topics tab brings to the web browser the ability to browse and interact with the Diffusion topic tree.

Copyright 2013 Push Technology

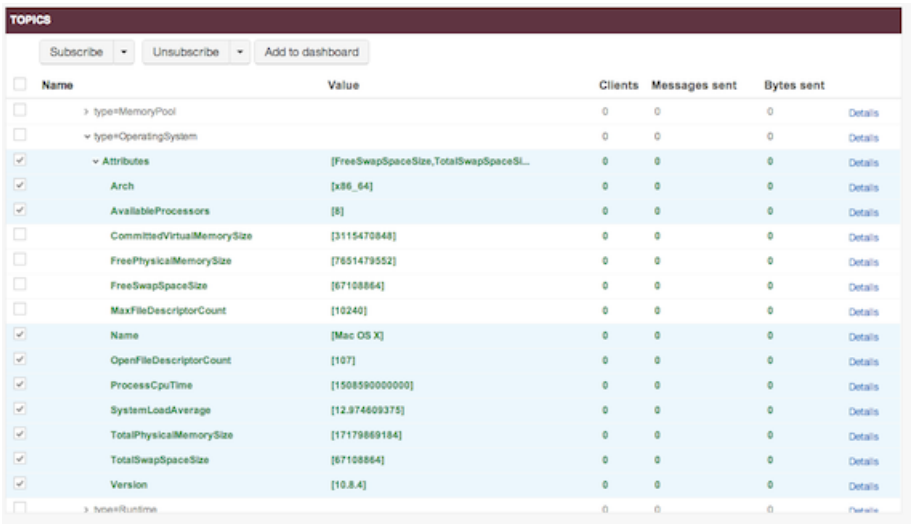


Figure 43: The table of Topics

Users may intuitively browse the live topic tree, fetch and subscribe to topics. If the server is so configured the table also shows the number of subscribed clients, messages sent and bytes sent. Enable individual topic statistics through `etc/Statistics.xml`, e.g.

```
<!-- Enable global topic statistics -->

  <topic-statistics enabled="true">

  <!-- Enable individual topic instance statistics -->

  <monitor-instances>true</monitor-instances>

</topic-statistics>
```

Once a set of topics is selected using its checkbox the Subscribe and Unsubscribe work intuitively, and each button has an recursive alternative available through the drop-down menu-button. “Add to dashboard” hints at it’s purpose and will be covered later on.

The details button shows more detail on the topic in question, as well as offering to fetch the topic value

Copyright 2013 Push Technology

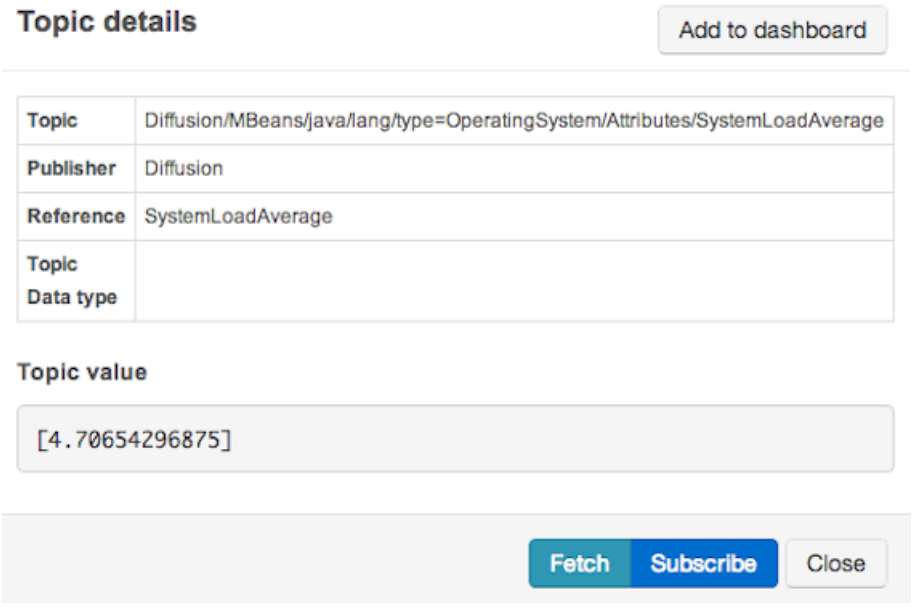


Figure 44: Details of the Topic publishing the CPU load of the host server

Features: Clients Tab

The Client tab shows a live list of the clients connected to the Diffusion server. Additionally it shows the number of messages to and from the server, the client IP address, connection type and connection time.

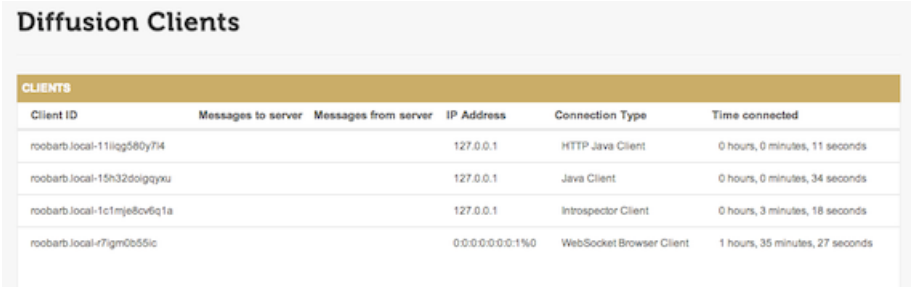


Figure 45: The table of Clients

Configure the Diffusion server to provide live client statistics through `etc/Statistics.xml`

```
<!-- Enable global client statistics -->
<client-statistics enabled="true">
  <!-- Definition of the log in Logs.xml -->
  <log-name>stats</log-name>
  <!-- Specifies the output frequency of the log, this is one entry per
 frequency -->

  <output-frequency>1h</output-frequency>
  <!-- Enable individual client instance statistics -->
  <monitor-instances>true</monitor-instances>
</client-statistics>
```

Features: Logs tab

The Logs tab shows a live colour-coded display of log entries emitted by the server up the level of **INFO**.

Copyright 2013 Push Technology

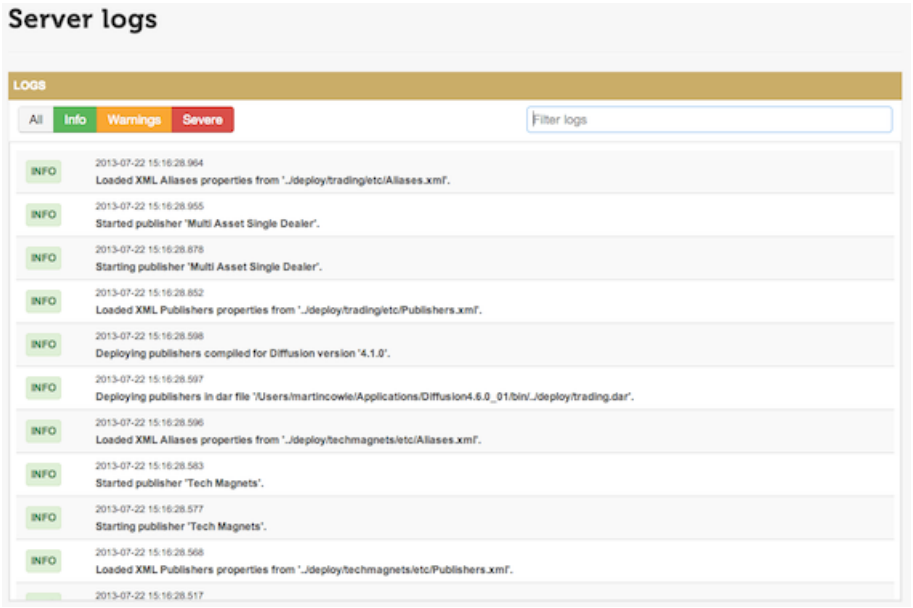


Figure 46: The table of log entries

User smay also perform client-side simple filtering on log entries. Unlike other monitoring metrics the Diffusion server retains up to 250 log entries in memory.

Advanced

Saving the console layout

Users can save changes made to their console layout with the “Save Dashboard layout” button. This persists a file on the server side, making it shared amongst all Console users.

White & Blacklist editing

The Console optionally maintains a blacklist or whitelist of IP addresses that are allowed to make use of it. Users may specifify discrete IP addresses or use syntax supported by *etc/SubscriptionValidationPolicy.xml* to cover subnets.

Figure 47: Editing the Access Policy

Stop Diffusion

The Stop Diffusion menu item will stop the server when clicked upon.

Copyright 2013 Push Technology

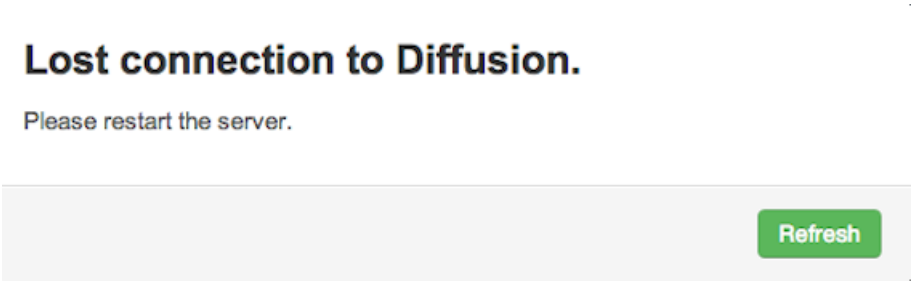


Figure 48: Notification that the Diffusion server has stopped

Going Further

Changing the console layout

The console is designed to be extensible and flexible. Users may reorder, edit, create and remove panels. Grab the panel header and drag it to a new location as desired. Click the trash-can icon to remove the icon - with verification

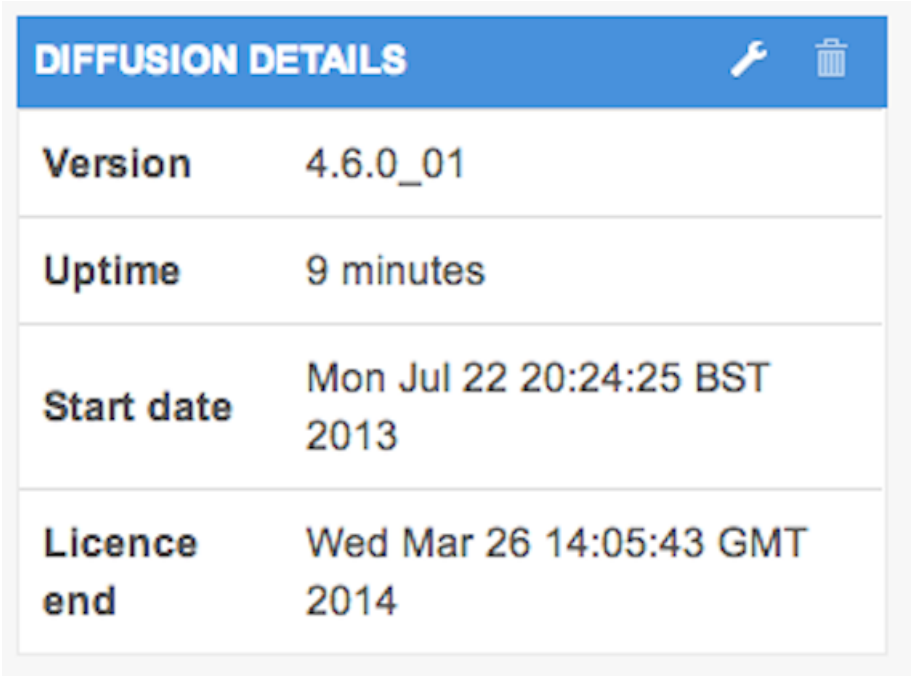


Figure 49: The default Diffusion Details panel

Users click on the spanner or wrench icon to configure the panel.

Copyright 2013 Push Technology

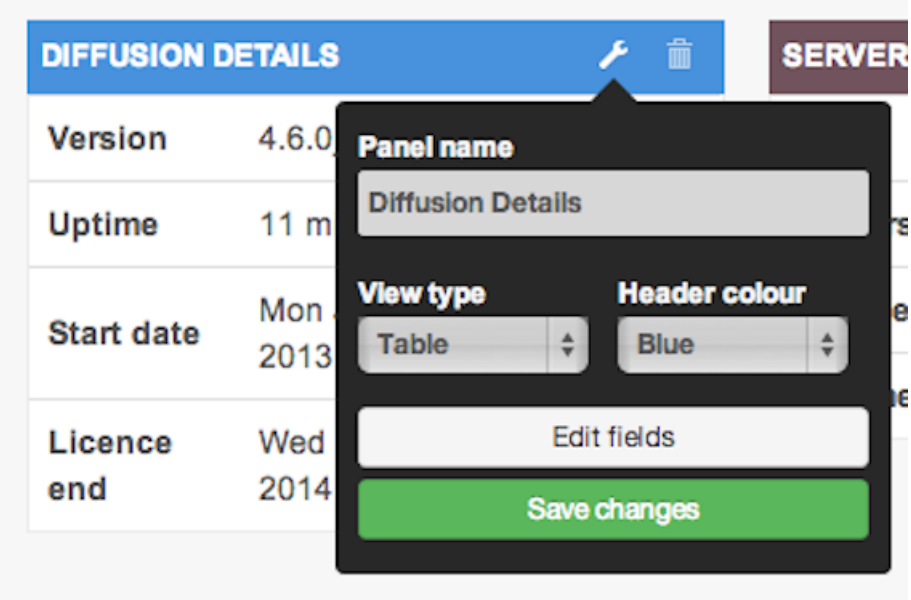


Figure 50: Editing the properties of the Diffusion Details panel

Panel name' and 'Header colour' are self explanatory. 'View type' is a choice of data renderings.

- **Table**: As seen already, this option shows one or more monitoring metrics in a table of textual values.
- **Line**: Renders one or more numeric metrics as a line graph.
- **Area**: Renders one or more numeric metrics as an overlapping area graph.
- **Single**: Used to visualise a single metric in large text, for metrics that are worth the screen real-estate.

Line and area graphs have an extra two configuration fields: 'Refresh rate (ms)' and 'Max data points'. The latter configures how much data is retained for rendering the graph. The former governs the frequency with which the graph is updated and does not influence the frequency of updates from the server. Historic data is only stored in the browser and refreshing the page will lose the stored set of data points.

Hovering the mouse over a graph panel will show the detail of the underlying data point

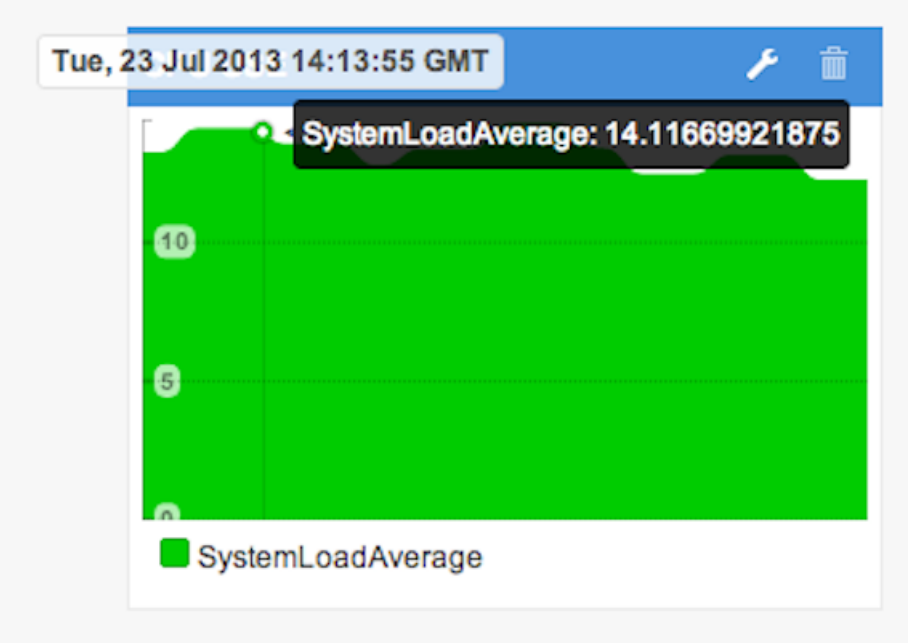


Figure 51: Visualising the CPU load on a server at a specific time.

Copyright 2013 Push Technology

Sourcing monitoring metrics

Clicking the 'Edit fields' button presents the user with a Topic Data Fields dialog, where the user nominates one or more topics from where metrics are drawn.

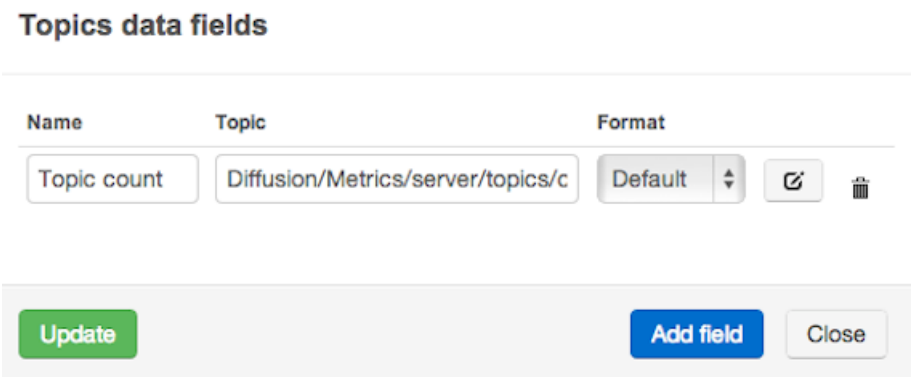


Figure 52: Editing and adding to the set of topics for this panel

Users of the Topics tab have already seen the "Add to dashboard" button in the "Topic details" dialog that can shortcut this process.

The default Console layout draws metrics from topics in the Diffusion/MBeans topic tree, however this is not mandatory and solution implementors are free to draw on any suitable topic to reflect their own monitoring needs - including 3rd party topics implemented as part of the solution.

The `Diffusion/MBeans` topic tree is populated by the JMX adapter which reflects all JMX MBeans as topics. Solution implementors that build custom MBeans to manage their solution can re-use the same MBeans for monitoring purposes.

The Console can draw on features that are themselves optional (Topic and Client statistics, for example). If they are disabled the Console will point this out, and request they be enabled in `etc/Statistics.xml`

Production deployment notes

Securing the `Diffusion/` topics

The topics in the `Diffusion/` tree convey a great deal of power and it is highly probable that bringing a Diffusion based solution to production requires limiting their access to suitable users: e.g. users with an IP address in a specific range. Solution implementors can achieve this by implementing an *auth-handler*.

The default configuration for the console allows users to stop and restart publishers as well as stop the Diffusion server itself. This feature is configured via the properties `console.control.server` and `console.control.publishers` on the Diffusion publisher in `etc/Publishers.xml`.

Basic integration with Splunk

How to achieve basic integration between Diffusion and the Splunk analysis and monitoring application

About

Splunk is a third party application from Splunk, Inc. which provides monitoring and analysis of other applications, primarily by parsing their logs and extracting information of interest. The information is then displayed through a web interface, which allows the creation of dashboards and alerts on user-defined events. Splunk is available for all major operating systems.

Diffusion's log format is designed to be consistent and to allow for easy parsing by monitoring tools, not limited to Splunk.

Copyright 2013 Push Technology

Installation

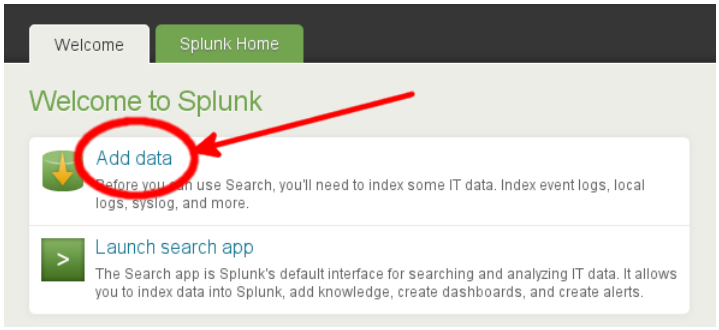
Installation typically takes just a few minutes, see the appropriate section of Splunk's *online documentation*. For simplicity, we will assume that Diffusion and Splunk are installed on the same machine.

Basic configuration

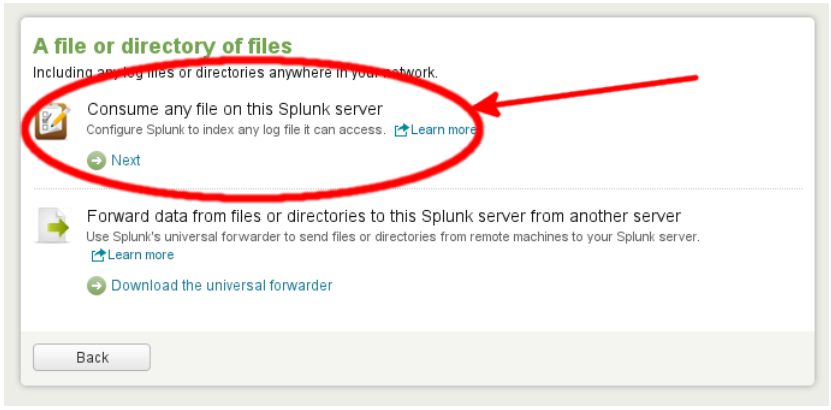
This is easier to do with existing log files to import, so configure Diffusion to write log files. To better demonstrate Splunk, set the server log file to `FINEST` logging in `etc/Logs.xml` and start Diffusion.

```
<!-- Example server log configuration -->
<log name="server">
 <log-directory>../logs</log-directory>
 <file-pattern>%s.log</file-pattern>
 <level>FINEST</level>
 <xml-format>false</xml-format>
 <file-limit>0</file-limit>
 <file-append>false</file-append>
 <file-count>1</file-count>
 <rotate-daily>false</rotate-daily>
</log>
```

On startup, access the Splunk web UI at *http://localhost:8000*. After logging in (and changing the default admin password), choose the “Add data” option.

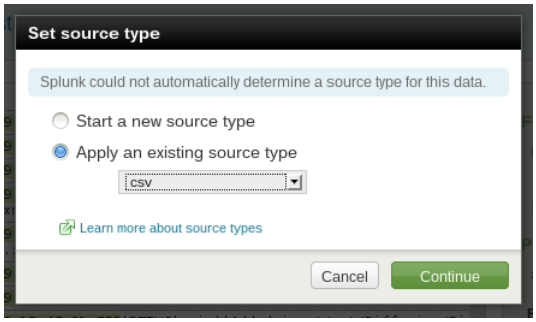

Welcome
Splunk Home
Welcome to Splunk
Add data
logs, syslog, and more.
Launch search app
The Search app is Splunk's default interface for searching and analyzing IT data. It allows you to index data into Splunk, add knowledge, create dashboards, and create alerts.

In the “Add Data to Splunk” screen that follows, choose the link “A file or directory of files” followed by “Consume any file on this Splunk server”.

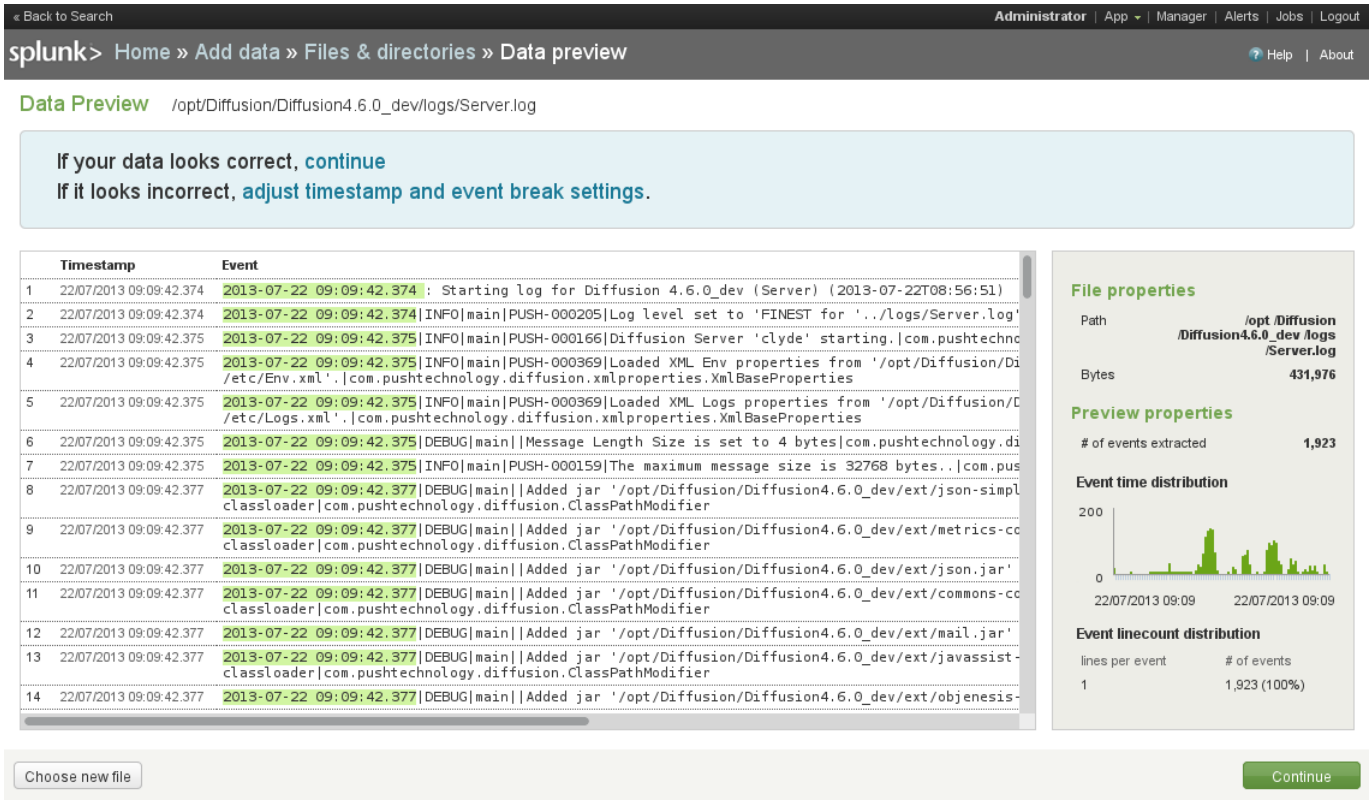

A file or directory of files
Consume any file on this Splunk server
Configure Splunk to index any log file it can access.
Learn more
Next
Forward data from files or directories to this Splunk server from another server
Use Splunk's universal forwarder to send files or directories from remote machines to your Splunk server.
Learn more
Download the universal forwarder
Back

Splunk may not be able to immediately identify the format of the log files; if this is the case then a dialog box similar to the following will be presented and you should select "csv" from the existing source types. Diffusion uses a pipe symbol rather than a comma as a separator but this is acceptible to Splunk's CSV parser.

Copyright 2013 Push Technology


Set source type
Splunk could not automatically determine a source type for this data.
Start a new source type
Apply an existing source type
csv
Learn more about source types
Cancel
Continue

The next dialog allows you to select Diffusion's `logs/Server.log` file under the "Preview data before indexing" option, which Splunk will read and parse. On the "Data Preview" screen, you should see numbered log entries with the timestamp highlighted; this indicates that the log file has been correctly parsed. Accept this, and on the next screen, set the source to be continuously indexing the data. You can leave the parameters in "More settings" at their default values. Once this is done, you've given the new data source a name (e.g. "Diffusion Server Log") and finally accepted the settings, you can begin searching and generating reports based on the log contents.

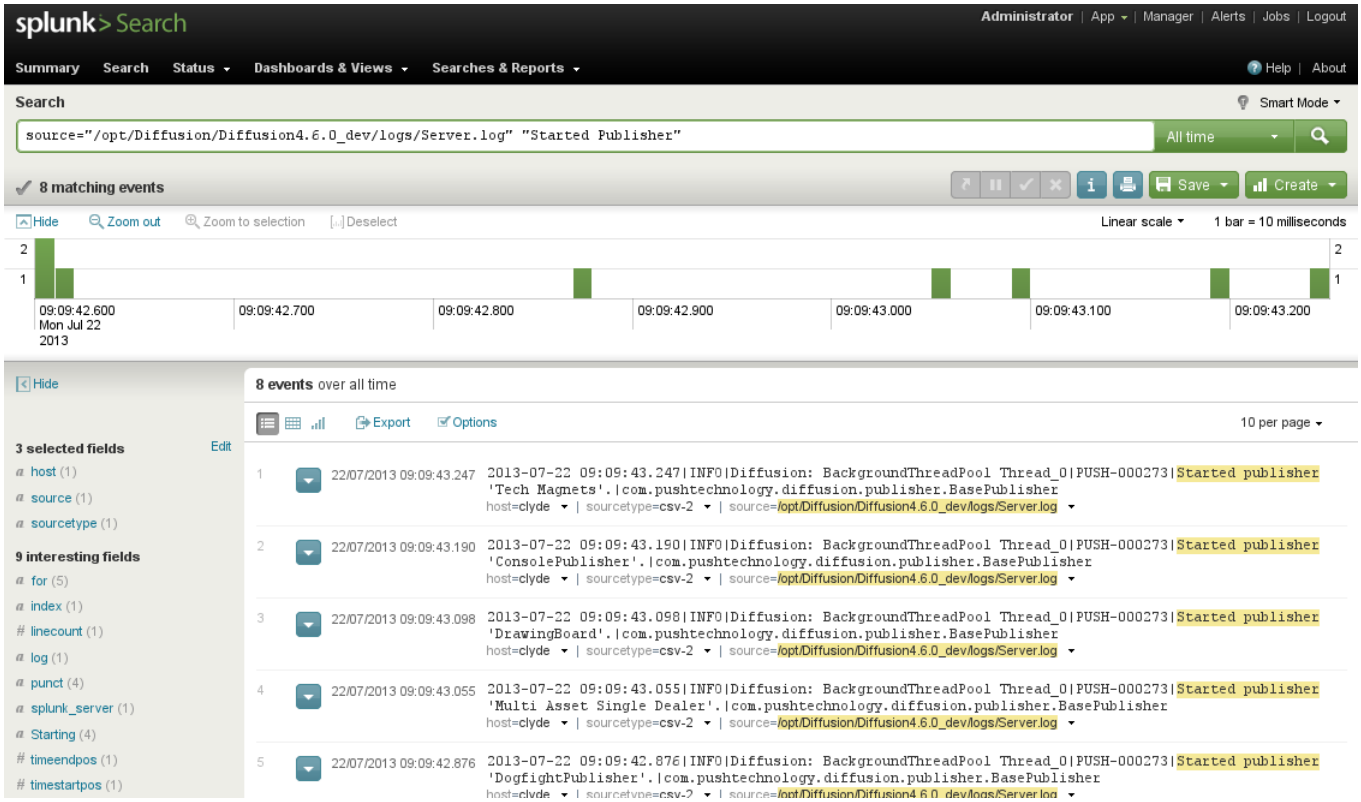

« Back to Search
Administrator | App | Manager | Alerts | Jobs | Logout
splunk> Home » Add data » Files & directories » Data preview
Help | About
Data Preview /opt/Diffusion/Diffusion4.6.0_dev/logs/Server.log
If your data looks correct, continue
If it looks incorrect, adjust timestamp and event break settings.
Timestamp Event
1 22/07/2013 09:09:42.374 2013-07-22 09:09:42.374 : Starting log for Diffusion 4.6.0_dev (Server) (2013-07-22T08:56:51)
2 22/07/2013 09:09:42.374 2013-07-22 09:09:42.374|INFO|main|PUSH-000205|Log level set to 'FINEST for '../logs/Server.log'
3 22/07/2013 09:09:42.375 2013-07-22 09:09:42.375|INFO|main|PUSH-000166|Diffusion Server 'clyde' starting.|com.pushtechno
4 22/07/2013 09:09:42.375 2013-07-22 09:09:42.375|INFO|main|PUSH-000369|Loaded XML Env properties from '/opt/Diffusion/Di /etc/Env.xml'.|com.pushtechnology.diffusion.xmlproperties.XmlBaseProperties
5 22/07/2013 09:09:42.375 2013-07-22 09:09:42.375|INFO|main|PUSH-000369|Loaded XML Logs properties from '/opt/Diffusion/D /etc/Logs.xml'.|com.pushtechnology.diffusion.xmlproperties.XmlBaseProperties
6 22/07/2013 09:09:42.375 2013-07-22 09:09:42.375|DEBUG|main||Message Length Size is set to 4 bytes|com.pushtechnology.di
7 22/07/2013 09:09:42.375 2013-07-22 09:09:42.375|INFO|main|PUSH-000159|The maximum message size is 32768 bytes..|com.pus
8 22/07/2013 09:09:42.377 2013-07-22 09:09:42.377|DEBUG|main||Added jar '/opt/Diffusion/Diffusion4.6.0_dev/ext/json-simpl classloader|com.pushtechnology.diffusion.ClassPathModifier
9 22/07/2013 09:09:42.377 2013-07-22 09:09:42.377|DEBUG|main||Added jar '/opt/Diffusion/Diffusion4.6.0_dev/ext/metrics-cc classloader|com.pushtechnology.diffusion.ClassPathModifier
10 22/07/2013 09:09:42.377 2013-07-22 09:09:42.377|DEBUG|main||Added jar '/opt/Diffusion/Diffusion4.6.0_dev/ext/json.jar'
11 22/07/2013 09:09:42.377 2013-07-22 09:09:42.377|DEBUG|main||Added jar '/opt/Diffusion/Diffusion4.6.0_dev/ext/commons-cc classloader|com.pushtechnology.diffusion.ClassPathModifier
12 22/07/2013 09:09:42.377 2013-07-22 09:09:42.377|DEBUG|main||Added jar '/opt/Diffusion/Diffusion4.6.0_dev/ext/mail.jar'
13 22/07/2013 09:09:42.377 2013-07-22 09:09:42.377|DEBUG|main||Added jar '/opt/Diffusion/Diffusion4.6.0_dev/ext/javassist- classloader|com.pushtechnology.diffusion.ClassPathModifier
14 22/07/2013 09:09:42.377 2013-07-22 09:09:42.377|DEBUG|main||Added jar '/opt/Diffusion/Diffusion4.6.0_dev/ext/objenesis-
File properties
Path /opt /Diffusion /Diffusion4.6.0_dev /logs /Server.log
Bytes 431,976
Preview properties
# of events extracted 1,923
Event time distribution
200
0
22/07/2013 09:09
22/07/2013 09:09
Event linecount distribution
lines per event # of events
1 1,923 (100%)
Choose new file
Continue

Simple searches

Now we have a data source configured, we can start to execute basic searches.

From the Splunk launch page, a "Search" option is now available. Selecting this will lead us to the Search Summary page, where you should select the Source relating to the file `logs/Server.log` previously imported. The page will change to include the source in the Search area; additional search terms can be added to the end e.g. "Started Publisher".

Copyright 2013 Push Technology



Further reading

To explore the full set of capabilities offered by Splunk, please refer to their documentation at *http://docs.splunk.com*.

Copyright 2013 Push Technology

Chapter 17

APIs

Topics:

- *Table of APIs*
- *Java API*
- *.Net API*
- *Javascript API*
- *ActionScript API*
- *Silverlight API*
- *iOS API*
- *Android API*
- *J2ME API*
- *C API*
- *DiffusionWrapper.js*

Diffusion™ provides a number of Application Programming Interfaces (APIs) which allow user written applications to make use of Diffusion™.

Where an API is not available for a particular language a user may still communicate with Diffusion™ via a TCP socket based connection using the *Diffusion™ Protocol*.

Table of APIs

	Publisher	Remote Service	Event Publisher	Client	Supported Version	Protocols
Java	*YES*	*YES*	*YES*	*YES*	Java 1.7	• DPT, DPTS • WS, WSS • HTTP, HTTPS (Full duplex)
.Net		*YES*	*YES*	*YES*	.Net 3.5	• DPT, DPTS • WS, WSS • HTTP (Full duplex)
JavaScript				*YES*	JavaScript V1.3	• Native: WS, WSS, HTTP, HTTPS • Flash: DPT, HTTP, HTTPS, HTTP Streaming • Silverlight: DPT, HTTP, HTTPS
Flash				*YES*	ActionScript V3.0 (Flex 3.0)	• DPT • HTTP, HTTPS • HTTPC, HTTPCS
Silverlight				*YES*	Silverlight V4.0	• DPT • HTTP, HTTPS • HTTPC, HTTPCS
iOS				*YES*	iOS V4.0	• DPT, DPTS
Android				*YES*	Android 2.2	• DPT, DPTS

Copyright 2013 Push Technology

Tier 2 APIs supported on a best effort basis

	Publisher	Remote Service	Event Publisher	Client	Supported Version	Protocols
J2ME				*YES*	Midlet V1.0	• DPT, DPTS
BlackBerry				YES	JRE 6.0	• DPT, DPTS
C				YES		• DPT

For more information on the transports supported by the clients please refer to the section *Transports Supported*

Table 49: Feature Matrix

Feature	Java	.NET	JavaScript	C	Flex	Silverlight	Java ME	iOS	Android
Asynchronous connect		YES	YES		YES	YES		YES	
'Simple' Connect	YES	YES	YES	YES	YES	YES	YES	YES	YES
Connect with topics (variable string args)	YES	YES	YES						
Connect with TopicSet	YES	YES							
Connect with multiple server details	YES	YES		YES	YES	YES	YES	YES	YES
Connection Cascading	YES	YES		YES	YES	YES	YES	YES	YES
Connection Failover	YES	YES			YES	YES	YES	YES	YES
Connection Load Balancing	YES	YES		YES	YES	YES	YES	YES	YES
Reconnect	YES	YES	YES	YES	YES	YES	YES	YES	YES
Get client Id	YES	YES	YES	YES	YES	YES	YES	YES	YES
'Is Connected' status	YES	YES	YES		YES	YES	YES	YES	YES

Copyright 2013 Push Technology

Feature	Java	.NET	JavaScript	C	Flex	Silverlight	Java ME	iOS	Android
'Is Reconnected' status	YES	YES	YES			YES	YES	YES	YES
Ping Server	YES	YES	YES	YES	YES	YES	YES	YES	YES
Create Topic Load message object	YES	YES					YES		YES
Create Delta message object	YES	YES	YES	YES	YES	YES	YES	YES	YES
Send (Topic Load) message	YES	YES	YES				YES		YES
Send (Delta) message	YES	YES	YES	YES	YES	YES	YES	YES	YES
(Auto) Message acknowledgement	YES	YES	YES	YES	YES	YES	YES	YES	YES
(Manual) Message acknowledgement	YES	YES	YES	YES	YES	YES	YES	YES	YES
Subscribe with topics (variable string args)	YES	YES					YES		YES
Subscribe with TopicSet	YES	YES							
Fetch with TopicSet	YES	YES							
Fetch	YES	YES	YES	YES	YES	YES	YES	YES	YES
Unsubscribe with topics (variable string args)	YES	YES							

Copyright 2013 Push Technology

Feature	Java	.NET	JavaScript	C	Flex	Silverlight	Java ME	iOS	Android
Unsubscribe with topic	YES	YES	YES	YES	YES	YES	YES	YES	YES
Unsubscribe with TopicSet	YES	YES							
Create Delta message	YES	YES							
Create Topic Load message	YES	YES							
Add Topic Listener	YES	YES	YES	YES	YES	YES	YES	YES	YES
Remove Topic Listener	YES	YES	YES	YES	YES	YES	YES	YES	YES
Add 'Timed' Topic Listener					YES				
'Read only' access into Topic listeners		YES							
Create Service Topic Handler	YES	YES	YES	YES	YES	YES	YES	YES	YES
Create Paged Topic Handler	YES	YES	YES	YES	YES	YES	YES	YES	YES
Create Topic Notify Topic Handler	YES	YES	YES		YES		YES	YES	YES
Fragmented messages	YES	YES	YES	YES	YES	YES	YES	YES	YES

Copyright 2013 Push Technology

Java API

The Java API comprises a number of packages all subordinate to the main `com.pushtechnology.diffusion.api` package.

Full Javadoc is issued with the product and therefore this section does not go into detail of how to use the API components. The following table is a guide to which packages contain the major API components.

Table 50: Java APIs

API Area	Relevant Packages
Publishers	The main classes for writing a Publisher are in the `publisher` package. General classes relating to Topic handling are in `topic` Classes related to Messages are in `message`. All classes relating to the use of TopicData are in the `data` package. Classes relating to the use of Conflation are in the `conflation` package. See *Publishers* for more information about how to use the Publisher API.
Client	The Client API classes are in the `client` package and makes use of the `connection` package for connection details. Client applications also make use of `config` for configuration and `message` for Message handling. The `threads` package also applies for thread pool handling.
Remote Control	The Remote Service API may be found in the `remote` package and it also uses the same common packages that the Client API uses. The server side of Remote Control is performed in Publishers using TopicData and the relevant classes can be found in the `data.remote` package. See *Remote Control* for more information about using the Remote Control feature.
Event Publisher	The Event Publisher API may be found in the `evpub` package and it also uses the same common packages that the Client API uses. See *Event Publishers* for more information about using an Event Publisher.
Server	Classes relating to the Diffusion™ Server (running it within an application) may be found in the `server` package and the configuration of the server would be done with the classes in the `config` package.
Web Server	Classes relating to implementing Diffusion™ Web Server functionality are in the `webserver` package.
Configuration	The `config` package contains all configuration classes used for configuring the Diffusion™ Server and/or within Client APIs.
System Management	System management classes may be found in the `management` package. Classes relating to statistics are in the `statistics` package.
WhoIs	If you want to implement your own WhoIs provider then the classes required are in the `whois` package.
General	General purpose classes relating to exceptions and logging as well as some useful utility classes may be found directly under the `api` package itself.

Copyright 2013 Push Technology

Client API

How to use the Java Client API

The Client API provides the ability to connect to a Diffusion™ Server as an External Client from within any Java application.

There is a single class called `ExternalClientConnection` which can be instantiated with the required connection details and then used to make an actual connection.

The connection class is of the generic type `ServerConnection` and as such, once a connection is made, any notifications or Messages from the Server are passed via the `ServerConnectionListener` interface.

The Topic(s) to subscribe to may be specified when connecting or at any time after connection.

The `ServerConnectionListener` specified will receive all Messages for all Topics. However, any number of additional Topic Listeners may be specified and Messages for different Topics routed to different listeners as required.

The API permits the following types of connection to be specified via the `ServerDetails` (see connection package) specified when configuring the connection object:-

TCP	For a standard connection over TCP/IP. This must connect to a standard Client Connector .
SSL	For a secure TCP/IP connection over SSL. This must connect to a Client Connector with SSL enabled
HTTP	For a connection using HTTP Protocol.
HTTP/SSL	For a connection using HTTP Protocol over SSL.

By specifying more than one `ServerDetails`, fallback connections can be specified, so that if the first connection does not succeed then the second is tried, and so on.

For a detailed description of the API see the issued Javadoc (in docs directory).

Authorisation Credentials

If authorisation credentials are required by the Diffusion™ Server then these are set at the `ConnectionDetails` level and used for all `ServerDetails` . Credentials may be set in a `ServerDetails` by creating a `Credentials` object and using `setCredentials` before connecting.

Credentials may also be sent to the Server after connection using the method `sendCredentials` in `ExternalClientConnection`. In this case it is possible that the Credentials will be rejected by the Server in which case this would be notified on the `serverRejectedCredentials` method of each `ServerConnectionListener` .

Reconnection

If a client unexpectedly loses connection then it can try to reconnect using the `reconnect` method. If the server has specified keep-alive for the connector then it may be possible for the client to pick up the same session as before and receive all messages that were queued for it whilst disconnected. The Topic state (i.e. the Topics subscribed to) will also be re-established on reconnection. If unable to reconnect then a new connection will be established with the same Topic set as used on the original connection. Successful reconnection or connection is notified on the normal `serverConnected` method and it is possible to determine which has occurred using the `ServerConnection.isReconnected()` method.

There is no guarantee that messages in transit at the time of the disconnection will be redelivered. However, all messages marked as requiring acknowledgement by the server will be delivered (i.e. no such messages would be lost)

Failover

The Java client supports autofailover. For more information you should refer to the *Java Failover* documentation.

Copyright 2013 Push Technology

Special Features

Paged Topic Data Handling

Where Paged Topic data is in use at the server there are features within the Client API which simplify the handling of messages to and from such a Topic.

Service Topic Data Handling

Where Service Topic data is in use at the server there are features within the Client API which simplify the handling of messages to and from such a Topic.

Example

The following example shows a simple Client class which sends a Message containing 'Hello' to the Server and logs all Messages it receives, until it receives a Message from the Server (Publisher) asking it to stop. It tries to connect via TCP first but if that fails it tries HTTP.

```
public class ClientApplication implements ServerConnectionListener {

    private static final Logger LOG =
        LoggerFactory.getLogger(ClientApplication.class);

    private ExternalClientConnection theConnection;

    public ClientApplication() throws APIException {
        // Create Connection
        theConnection=
            new ExternalClientConnection(
                this,
                "dpt://localhost:8080",
                "http://localhost:8080");
        // Connect, subscribing to a single topic
        theConnection.connect("MyTopic");
        // Send a Message
        TopicMessage message = theConnection.createDeltaMessage("MyTopic");
        message.put("Hello");
        theConnection.send(message);
    }

    public void messageFromServer(
    ServerConnection serverConnection,
    TopicMessage message) {
        LOG.info("Message Received : {}",message);
        try {
            if (message.asString().equals("STOP")) {
                theConnection.close();
            }
        }
        catch(Exception ex) {
            ex.printStackTrace();
        }
    }

    public void serverConnected(ServerConnection serverConnection){
        LOG.info("Connected to Server : {}",serverConnection);
    }

    public void serverTopicStatusChanged(
    ServerConnection serverConnection,
    String topicName,
    TopicStatus status) {
        LOG.info(
```

Copyright 2013 Push Technology

```
            "Topic {} at {} status changed to {}",
            topicName,serverConnection,status);
    }

    public void serverRejectedCredentials(
    ServerConnection serverConnection,
    Credentials credentials) {
        LOG.info("Server Rejected Credentials : {}",serverConnection);

    }

    public void serverDisconnected(ServerConnection serverConnection) {
        LOG.info("Disconnected from Server : {}",serverConnection);
    }

}
```

Event Publisher API

How to use the Java Event Publisher API

The main class is `EventPublisherConnection` which is instantiated to configure the connection and then `connect` is called to make an actual connection.

Messages may then be sent to the Server and each will be routed to the Publisher that owns the Message Topic. The Server can also send Messages to the Event Publisher.

The connection may only be made over DPT and as such `ServerDetails` may not be specified but may be obtained to change details before connection.

The port specified to the connection must correspond to an Event Publisher Connector for a Diffusion™ Server running on the specified host.

For a detailed description of the API see the issued Javadoc (in docs directory).

The following example shows the simple case of establishing a connection and sending a single Message, though in reality an Event Publisher would send many Messages over a period of time.

```
public class EventPublisherApplication implements EventPublisherListener {

    private static final Logger LOG =
        LoggerFactory.getLogger(EventPublisherApplication.class);

    private EventPublisherConnection theConnection;

    public EventPublisherApplication() throws APIException {
        theConnection=new EventPublisherConnection("localhost",3098,this);
        theConnection.getServerDetails().setOutputBufferSize(8000);
        theConnection.connect();
        TopicMessage message =
 theConnection.createDeltaMessage("MyTopic",0);
        theConnection.send(message);
    }

    public void messageFromServer(
    EventPublisherConnection connection,
    TopicMessage message) {
        LOG.info(
            "Event Publisher Connection {} received message from server :
 {}",
            connection,message);
    }
```

Copyright 2013 Push Technology

```
    public void serverDisconnected(EventPublisherConnection connection) {
        LOG.info(
            "Event Publisher Connection Closed {}",connection);
    }

}
```

Remote Service API

How to use the Java Remote Service API

A Remote Service is provided by a Java application that connects to a Diffusion™ Server implementing some form of Remote Topic Data which provides the server side functionality to support Remote Services.

See *Remote Control* for general details of how to use the Remote Control features.

Instantiating a Remote Service

To instantiate a `RemoteService` object the `RemoteServiceFactory` class is used.

For example:-

```
ServerDetails serverDetails =
           ConnectionFactory.createServerDetails("dpt://localhost:8080");

   serverDetails.setInputBufferSize(128*1024);

   RemoteService remoteService =
       RemoteServiceFactory.createRemoteService(
           serverDetails,
           "RemoteControl",
           "MyDomain",
           this); // where this is a RemoteServiceListener
```

The `ServerDetails` specifies the location of the Diffusion™ Server to connect to.

The name of the Remote Control Topic (" `RemoteControl` " in the example) on the Server must be specified. This Topic must already exist on the Server when the remote service registers.

The name of the Domain Topic (" `MyDomain` " in the example) to be created at the Server must be specified. This Topic must not already exist on the Server when the remote service registers. All of the Topics that the Remote Service creates and manages will be subordinate to this Topic.

Finally a listener for notifications from the Server needs to be specified. This must implement the interface `RemoteServiceListener` the methods of which are listed in Service Notifications .

Once a `RemoteService` has been created then it may be used to register the Remote Service with the Server.

Once a `RemoteService` has been registered it may be used to issue commands to the Server using the Service methods .

Setting Environmental Properties

In the above example the input bufer size on the server details is seen as being set. There are various environmental properties that may need to be set in order to get the best performance from a Remote Service or to tune it to user requirements. Some examples of these are shown in the table below:-

Copyright 2013 Push Technology

Table 51: Environment Properties

Property	Set on	Description
Input Buffer Size	`ServerDetails`	This specifies the size of the local socket input buffer to be used for receiving messages from the server. It should correspond to the output buffer size set on the Event Publisher connector being connected to on the server.
Output Buffer Size	`ServerDetails`	This specifies the size of the local socket output buffer used for sending messages to the server. It should correspond to the input buffer size on the Event Publisher connector at the server.
Message Queue Size	`RemoteService`	The underlying Event Publisher conection uses a local queue for outbound messages. If the throughput is very high from the Remote Service then this value may need to be tuned.
General API Properties	`ConfigManager`	There are a number of general API level properties, such as maximum message size and logging levels, that may be set using the `ConfigManager` interface - see Javadoc for details.

Service Options

Certain options may be set which change the default behaviour of the Remote Service. These options must be set before registering the service otherwise they will be ignored.

Options are set by getting the `RemoteServiceOptions` object from the `RemoteService` using its `getOptions()` method and then setting options as appropriate.

Options are as follows:-

Table 52: Service Options

Option Method	Description
`setClientConnectNotifications`	By default a Remote Service will only receive notifications of Clients which make requests of Topics within the Remote Service's domain. The Client Notifications option allows notifications of all Client connections to be routed to the Remote Service.
`setAuthoriseSubscribeClients`	By default when the `subscribeClients` method is called then all Clients are subscribed immediately on the Server to the selected Topics without sending `clientSubscribe` notifications back to the Remote Service. If this option is set then subscriptions will not occur immediately but `clientSubscribe` notifications will be sent back to the Remote Service for each Client/Topic subscription selected.
`setRouteSelectorSubscribes`	By default if a Client subscribes to a selector pattern such as `Domain/` or `Domain/foo/` then the Remote Service will only receive `clientSubscribe` notifications for any existing subdomain Topics that satisfy the selector pattern. If this option is set then the Remote Service will be notified of such selector subscription attempts on the `RemoteServiceListener.clientSubscribe(ClientDetails,TopicSelecto` method rather than attempting to resolve any such selection on the server.

Copyright 2013 Push Technology

For example:-

```
RemoteServiceOptions options = remoteService.getOptions();
options.setClientConnectNotifications(true);
options.setRouteSelectorSubscribes(true);
```

Registering a Remote Service

A `RemoteService` may be registered with the specified Server using the register method, for example:-

```
remoteService.register();
```

This causes a connection to be made to the Server and a registration message sent to the specified Control Topic at the Server.

The specified Domain Topic will then be created at the Server and a notification sent back on the `RemoteServiceListener.registered` method to indicate that the service was registered.

A notification will also be sent back on the `registerFailed` method if the registration fails.

Service Methods

The following tables lists the main methods on the `RemoteService` object which can be used for communicating with the Server.

For full details of the API see the issued Javadoc.

These methods may not be called until the `RemoteServiceListener` has received notification that it has successfully registered.

Table 53: Service Methods

Method	Description
`addTopic`	This is used to send a request to the Server to create a new Topic. A `TopicSpecification` must be specified which defines the Topic type and and any additional information required (such as metadata).
`sendMetadata`	Allows a metadata definition to be sent that can be referenced by name in future calls to `addTopic`
`closeClient`	This is used to close a specified Client. This will send a message to the Server requesting that the Client (at the edge) be disconnected.
`publish`	This is used to publish a message to all Clients subscribed to a Topic. The Topic must be one created within the Service Domain.
`removeTopic`	This is used to remove a (subdomain) Topic from the Server.
`sendFetchReply`	This is used to return a reply to a fetch request as received on the `clientFetch` method of the `RemoteServiceListener` . The data to include in the fetch reply can be sent but if it is not the data can be obtained from the Topic in the (edge) Server.
`sendToClient`	This is used to send a message to a specific Client. The message must be sent on a subdomain Topic.
`sendToPublisher`	This is used to send a Message to the Edge Publisher. The message will be received at the Publisher on its `RemoteControlTopicDataListener.messageFromService` method.
`subscribeClient`	This is used to subscribe a Client to a specified Topic. The Topic that is subscribed must be a subdomain Topic. If the Topic is a routing Topic then a target Topic must also be specified. A Topic selector may be used to subscribe multiple Topics.

Copyright 2013 Push Technology

Method	Description
`subscribeClients`	This is used to send a request to the Edge Publisher to subscribe all (or only those that have pre-emptively subscribed) Clients to one or more subdomain Topics.
`unregister`	This is used to unregister the service and close the connection. When a service is unregistered it"s Domain Topic (and all subdomain Topics) is automatically removed at the Server so there is no need to remove individually.
`unsubscribeClient`	This is used to unsubscribe a specified Client from a subdomain Topic. A Topic selector may be used to unsubscribe multiple Topics.

Client Details

A `RemoteService` maintains a map of all of the Clients that are connected to it therefore there is no need for the Remote Service application to do the same.

Methods such as `getClientDetails` and `getClients` are available to access the map of Clients.

Service Notifications

When a `RemoteService` is created a `RemoteServiceListener` is nominated to receive notifications from the Server.

The methods on this interface are listed below but the Javadoc should be consulted for full details.

Table 54: Service Notifications

Notification Method	Description
`clientConnected`	Provides details of a Client that has connected at the edge Server. By default this will be notified before the first subdomain Topic related notification from the Client. However, if the client notifications option has been set before registration this will be notified for every Client that connects.
`clientDetailsChanged`	A Client that has been previously notified to the Service has changed its credentials.
`clientDisconnected`	A Client that has previously been notified to the Service has (or has been) disconnected from the edge Server.
`clientFetch`	A Client has requested a 'fetch' of a subdomain Topic (which could be a Topic that does not exist). Assuming the fetch is permitted the reply to the request may be sent on the `sendFetchReply` method of the `RemoteService`
`clientSubscribe`	There are two variants of this notification - one passing a String topic name and one passing a Topic Selector. The String variant notifies that a Client has requested subscription to a subdomain Topic (which could be a Topic that does not exist). Assuming the subscription is permitted then it may be actioned using one of the `subscribeClient` methods on the `RemoteService` . If the Topic does not exist it may be necessary to create it first. The Topic Selector variant is notified when the `setRouteSelectorSubscribes` option is chosen and a Client attempts to subscribe to a selector pattern that selects children of possible Domain Topics (e.g.Domain/ or Domain/foo//).
`closed`	The service has been closed. Note that when a service is closed then its Domain Topic (and all subdomains) will automatically be removed at the server.
`messageFromClient`	A Client has sent a message to a subdomain Topic. The message content will be provided.
`messageFromPublisher`	Receives a Message that has been sent from the Edge Publisher (as opposed to from a Client).

Copyright 2013 Push Technology

Notification Method	Description
`registered`	The service has been successfully registered. This must be received before the `RemoteService` methods may be used to send requests to the Server.
`registerFailed`	The registration of the service has failed.
`serviceRequest`	A Client has sent a request to a subdomain Topic which has Service Topic Data. The request content is provided in the form of a `RemoteRequest` object which may be used to return a reply to the request.
`topicAddFailed`	A request to add a Topic (`RemoteService.addTopic`) has failed.
`topicSubscribeFailed`	A request to subscribe a Client to a Topic (`RemoteService.subscribeClient`) has failed.

Using Paged Topics

Topics that are 'Paged' are different from other Topics in the sense that there are some special handling mechanisms in the Remote Service that allow the paged Topic to be updated.

A paged topic is created by a Remote Service in the same way as other Topics, using a `TopicSpecification` object.

The following shows an example of the creation and use of a paged string topic:-

```
        PagedStringTopicSpecification spec = new
PagedStringTopicSpecification();

        theRemoteService.addTopic("PagedTopic",spec);

        PagedStringTopicUpdater updater =
            spec.createUpdater("PagedTopic",theRemoteService);

        updater.add("line 1");
```

The significant part is the creation of the updater object which is what will be used to maintain the paged data. The example shows a single line being added to the data but there are methods on the updater that provide all of the updating options available to *Paged Topic Data* when used within a Publisher.

The example shows the creation of paged string topic data but paged record topic data can be created in a similar way using a `PagedRecordTopicSpecification`. This would then be able to return a `PagedRecordTopicUpdater`.

The example shows the creation of unordered paged data but ordered paged data can also be created as follows:-

```
        PagedStringTopicSpecification spec = new
PagedStringTopicSpecification();

        spec.setOrdered("com.acme.MyComparator",Duplicates.FIRST);
```

Here the name of a Comparator class that may be instantiated at the server must be specified.

Example Remote Service Application

Below is an example of a class that acts as a simple Remote Service application.

When the 'start' method is invoked it connects and registers and then when registration is complete it creates two topics called 'clients' and 'service'.

The 'clients' topic notifies all subscribed clients whenever any client connects, subscribes (to 'clients'), unsubscribes or disconnects. When a client subscribes to the topic it gets a load message containing all connected clients. The service

Copyright 2013 Push Technology

will also return all of the clients to any client that does a 'fetch' on the Topic. Clients are only able to subscribe to this Topic if they provide a password in their credentials.

The 'service' Topic provides a simple request/response service which has a single function, thet being to get the details (user name and remote address) of a specified client.

This class is an example and does not perform any error handling.

```
public class RemoteServiceApplication implements RemoteServiceListener {

    private static final Logger LOG =
        LoggerFactory.getLogger(RemoteServiceApplication.class);

    private static final String PASSWORD = "servicePassword";
    private static final String CLIENTS_TOPIC = "clients";
    private static final String SERVICE_TOPIC = "service";
    private static final String CLIENT_ADDED = "A";
    private static final String CLIENT_SUBSCRIBED = "S";
    private static final String CLIENT_UNSUBSCRIBED = "U";
    private static final String CLIENT_CLOSED = "C";

    private RemoteService theRemoteService;

    void start() throws APIException {

        // Create remote service
        ServerDetails serverDetails =
            ConnectionFactory.createServerDetails("dpt://localhost:8080");

        serverDetails.setInputBufferSize(128*1024);

        RemoteService remoteService =
            RemoteServiceFactory.createRemoteService(
                serverDetails,
                "RemoteControl",
                "MyDomain",
                this);

        // Set Service Options
        RemoteServiceOptions options = remoteService.getOptions();
        options.setClientConnectNotifications(true);
        options.setRouteSelectorSubscribes(true);

        theRemoteService=remoteService;

        // Connect and register
        remoteService.register();

    }

    @Override
    public void registered() {
        // Create a 'clients' topic and a 'service' Topic
        try {
            TopicSpecification spec = new SimpleTopicSpecification();
            theRemoteService.addTopic(CLIENTS_TOPIC,spec);

            spec = new ServiceTopicSpecification("SV");
            theRemoteService.addTopic(SERVICE_TOPIC,spec);
        }
        catch (APIException ex) {
            ex.printStackTrace();
        }
    }
```

Copyright 2013 Push Technology

```
   @Override
   public void registerFailed(String errorMessage) {
       LOG.error("Registration failed with : {}",errorMessage);
   }

   @Override
   public void clientConnected(ClientDetails clientDetails) {
       // Notify all subscribed clients that a new client has connected
       publishClientsMessage(CLIENT_ADDED,clientDetails.getClientID());
   }

   @Override
   public void clientDetailsChanged(ClientDetails clientDetails) {
       // No action
   }

   @Override
   public void clientSubscribe(ClientDetails clientDetails,String
topicName) {
       // We only allow a client to subscribe to one of our Topics if they
       // have provided a password
       if (clientDetails.getCredentials().getPassword().equals(PASSWORD)) {
           String clientId = clientDetails.getClientID();
           try {
               // Perform the subscription
               theRemoteService.subscribeClient(clientId,topicName);

               if (topicName.equals(CLIENTS_TOPIC)) {
                   // Notify all subscribed clients that a new client has
subscribed
                   publishClientsMessage(CLIENT_SUBSCRIBED,clientId);

                   // And send a Topic Load to the subscribing client

theRemoteService.sendToClient(clientId,createClientsTopicLoad());
               }
           }
           catch (APIException ex) {
               ex.printStackTrace();
           }
       }
   }

   @Override
   public void clientSubscribe(ClientDetails clientDetails,TopicSelector
selector) {
       LOG.warn(
           "Client {} attempted to subscribe using
{}",clientDetails,selector);
   }

   @Override
   public void clientFetch(ClientDetails clientDetails,String topicName,
   List<String> headers) {
       if (topicName.equals(CLIENTS_TOPIC)) {
           // Send a topic load as fetch reply
           try {
               theRemoteService.sendFetchReply(
                   clientDetails.getClientID(),
                   createClientsTopicLoad(),
                   headers);
           }
           catch (APIException ex) {
```

Copyright 2013 Push Technology

```
                ex.printStackTrace();
            }
        }
    }

    @Override
    public void clientUnsubscribe(String clientId,String topicName,
    boolean hasSubscribers) {
        // Notify all subscribed clients that a new client has unsubscribed
        if (topicName.equals(CLIENTS_TOPIC)) {
            publishClientsMessage(CLIENT_UNSUBSCRIBED,clientId);
        }
    }

    @Override
    public void clientDisconnected(String clientId) {
        // Notify all subscribed clients that a new client has disconnected
        publishClientsMessage(CLIENT_CLOSED,clientId);
    }

    @Override
    public void topicAddFailed(String topicName,String errorMessage) {
        LOG.warn("Topic add failed {} : {}",topicName,errorMessage);
    }

    @Override
    public void topicSubscribeFailed(String clientId,String topicName,
    String errorMessage) {
        LOG.warn("Topic subscribe failed {} : {}",topicName,errorMessage);

    }

    @Override
    public void messageFromClient(ClientDetails clientDetails,String
topicName,
    TopicMessage message) {
        // Not implemented
    }

    @Override
    public void messageFromPublisher(TopicMessage message) {
        // Not implemented
    }

    @Override
    public void serviceRequest(RemoteRequest request) {
        if (request.getRequestType().equals("getClientDetails")) {
            try {
                // Return details of client specified in request
                String clientId = request.getRequestMessage().asString();
                ClientDetails client =
theRemoteService.getClientDetails(clientId);
                if (client!=null) {
                    // Create a response message
                    TopicMessage response =

theRemoteService.createDeltaMessage(SERVICE_TOPIC,100);
                    // Put the client details into it
                    response.putFields(
                        clientId,
                        client.getCredentials().getUsername(),
                        client.getRemoteAddress());
                    // And send the response back to the client
```

Copyright 2013 Push Technology

```
request.serviceResponse(request.getRequestType(),response);
                }
            }
            catch (APIException ex) {
                ex.printStackTrace();
            }
        }
    }

    @Override
    public void closed(RemoteServiceCloseReason reason) {
        LOG.info("Remote Service Closed : {}",reason);
    }

    /**
     * Creates a Topic Load for the 'clients' topic.
     */
    private TopicMessage createClientsTopicLoad() throws APIException {
        TopicMessage message =
theRemoteService.createLoadMessage(CLIENTS_TOPIC,1000);
        for (ClientDetails client:theRemoteService.getClients()) {
            message.putFields(client.getClientID());
        }
        return message;
    }

    /**
     * Publish a notification on the 'clients' topic.
     */
    private void publishClientsMessage(String header,String clientId) {
        TopicMessage message;
        try {
            message =
theRemoteService.createDeltaMessage(CLIENTS_TOPIC,100);
            message.setHeaders(header);
            message.put(clientId);
            theRemoteService.publish(message);
        }
        catch (APIException ex) {
            ex.printStackTrace();
        }
    }


}
```

.Net API

The .Net API comprises a number of packages.

Client API

How to use the .NET Client API

The ExternalClient API provides the ability to connect to a Diffusion™ Server as an External Client from within any .NET application.

There is a single class called `ExternalClient` which can be instantiated with the required connection details and then used to make an actual connection.

The Topic(s) to subscribe to may be specified when connecting or at any time after connection.

Copyright 2013 Push Technology

When a connection object is instantiated, the `InboundMessageReceived` delegate should be subscribed to, which will receive all Messages for all Topics.

The API permits the following types of connection to be specified via the `ServerDetails` specified when configuring the connection object:-

TCP	For a standard connection over DPT. This will connect to the Java External Client Connector.
SSL	For a secure TCP/IP connection over DPTS. This will connect to the Java External Client Connector.
HTTP	For a connection using HTTP Protocol

For a detailed description of the API, see the issued documentation (in docs directory).

The following example shows a simple Client class which sends a Message containing 'Hello' to the Server until it receives a message from the Server (Publisher) asking it to stop.

```
public class ClientApplication : IServerConnectionListener, ITopicListener
{
 #region Fields

 private readonly PushTechnology.DiffusionExternalClient.ExternalClient
theClient;

 #endregion // Fields

 #region Constructor

 public ClientApplication()
 {
  var connectionDetails =
   ConnectionFactory.CreateConnectionDetails( "dpt://localhost:8080",
"http://localhost:8080" );

  connectionDetails.Topics = new TopicSet("MyTopic");

  theClient = new
PushTechnology.DiffusionExternalClient.ExternalClient(connectionDetails);

  // Add a topic listener - we're listening to all messages for this
example, but individual topics can
  // also be used as selectors
  theClient.AddGlobalTopicListener( this );

  // Now connect - this is an asynchronous process, so we have to wait
until 'ServerConnected' is invoked
  theClient.Connect();
 }

 #endregion // Constructor

 #region Implementation of IServerConnectionListener

 /// <summary>
 /// Notification of connection.
 ///
 /// This is called when a connection to a server is established.
 /// </summary>
 /// <param name="connector">The server connector.</param>
 public void ServerConnected( IDiffusionClientConnector connector )
 {
```

Copyright 2013 Push Technology

```
   Console.WriteLine( "Connected to server: " + connector );

   // Send a message as we're now connected
   ITopicMessage message = theClient.CreateDeltaMessage( "MyTopic" );

   // Populate the message
   message.Put( "Hello" );

   // Send the message to the Diffusion server
   theClient.SendMessage( message );
  }

  /// <summary>
  /// Notification that the status for a topic that was subscribed to has
changed.
  /// </summary>
  /// <param name="connector">The connector.</param>
  /// <param name="topicName">The name of the topic on which the status has
changed.</param>
  /// <param name="statusType">The topic status change type.</param>
  public void ServerTopicStatusChanged( IDiffusionClientConnector connector,
string topicName, TopicStatusChangeType statusType )
  {
   Console.WriteLine(
    string.Format( "Topic status for '{0}' changed to '{1}'.", topicName,
statusType ));
  }

  /// <summary>
  /// Notification of rejected credentials from the server.
  /// </summary>
  /// <param name="connector"></param>
  /// <param name="credentials"></param>
  public void ServerRejectedCredentials( IDiffusionClientConnector
connector, Credentials credentials )
  {
   Console.WriteLine( "Server rejected credentials.");
  }

  /// <summary>
  /// Notification of disconnection.
  ///
  /// The reason for the disconnection can be established by checking the
state of the connection
  /// using IDiffusionClientConnector.State.
  /// </summary>
  /// <param name="connector">The server connector.</param>
  /// <param name="args">The arguments which can be interrogated for the
state and details of a server closure.</param>
  public void ServerDisconnected( IDiffusionClientConnector connector,
ServerClosedEventArgs args )
  {
   Console.WriteLine( "Disconnected from server.");
  }

  #endregion

  #region Implementation of ITopicListener

  /// <summary>
  /// Handles a message received from an IMessageSource.
  ///
  /// This will handle an incoming message from a specified source.
  /// </summary>
```

Copyright 2013 Push Technology

```
 /// <param name="source">The message source.</param>
 /// <param name="message">The message.</param>
 public bool HandleTopicMessage( IMessageSource source, ITopicMessage
message )
 {
  if (message.AsString().Equals("STOP"))
  {
   theClient.Disconnect();
  }

  return false;
 }

 #endregion
}
```

Event Publisher API

How to use the .NET Event Publisher API

The ExternalClient API provides the ability to connect to a Diffusion™ Server as an External Client from within any .NET application.

There is a single class called `EventPublisher` which can be instantiated with the required connection details and then used to make an actual connection. Messages may then be sent to the Server and each will be routed to the Publisher that owns the Message Topic.

The connection may only be made over DPT and as such `ServerDetails` may not be specified.

The port specified to the connection must correspond to an Event Publisher Connector for a Diffusion™ Server running on the specified host.

For a detailed description of the API see the issued documentation (in docs docs directory).

The following example shows the simple case of establishing a connection and sending a single Message, though in reality an Event Publisher would send many Messages over a period of time.

```
publicclassEventPublisherApplication : IEventPublisherListener
  {
       #region Fields
       privatereadonlyEventPublisherthePublisher;
       #endregion// Fields
       #region Constructor
       publicEventPublisherApplication()
       {
           // Instantiate the Event Publisher
           thePublisher =
               newEventPublisher( "localhost", "3098", this );
           thePublisher.Connect();
       }
       #endregion// Constructor
       ///<summary>
       /// Notification of connection.
       ///
       /// This is called when a connection to an Event Publisher is
established.
       ///</summary>
       ///<param name="connector">The server connector.</param>
       publicvoidServerConnected(DiffusionEventPublisherConnectorconnector)
       {
           ITopicMessagemessage =
               connector.CreateDeltaMessage( "MyTopic" );
```

Copyright 2013 Push Technology

```
            connector.Send( message );
        }
        publicvoidServerDisconnected(
            DiffusionEventPublisherConnectorconnector,
            ServerClosedEventArgsargs )
        {
            Debugging.StackTraceDebugPrint(
                "Event Publisher connection closed: " + connector );
        }
        publicvoidMessageFromServer(
            DiffusionEventPublisherConnectorconnector,
            MessageImplmessage )
        {
            Debugging.StackTraceDebugPrint(
                "Message from server: " + message );
        }
    }
```

Remote Service API

How to use the .NET Remote Service API

`PushTechnology.DiffusionRemoting`

This assembly provides the classes necessary to implement a *Remote Service* in .NET. A Remote Service is provided by a .NET application that connects to a Diffusion Server implementing some form of Remote Topic Data which provides the server side functionality to support Remote Services.

This is effectively the same as the *Java API*. See issued API documentation for full details.

Javascript API

The Javascript API provides *modern web developers* with a simple means of connecting to and interacting with a Diffusion server. The API takes care to select the most appropriate underlying transport from those available.

Using the JavaScript API

To enable the Diffusion Client `diffusion.js` needs to be loaded by the web page.

```
<script type="text/javascript" src="lib/DIFFUSION/diffusion.js"></script>
```

The Diffusion Client also needs a element with an id of *DiffusionContainer* and example is listed below.

```
<div id="DiffusionContainer">
 <script type="text/javascript">
  DiffusionClient.connect( { topic : "Chat,Subscribers", onDataFunction :
 onDataEvent });
 </script>
</div>
```

As you can see, it is possible to embed the DiffusionClient within the *DiffusionContainer* although not a requirement.

Connecting

Diffusion Client is a singleton with global scope and can be called from anywhere. To connect to Diffusion™, you need to call the connect method. The method takes two parameters, first is the the connection object and the optional second object is the client credentials. The example above is using an anonymous connection object.

Credentials

Copyright 2013 Push Technology

Credentials can either be supplied on the connect method, or set separately using the `DiffusionClient.setCredentials(..)` method. These credentials will be used for all transports that are used to connect to Diffusion™. The `DiffusionClientCredentials` object is a simple one of username and password attributes.

Connection Object

The Connection Object has over twenty attributes, that vary the way that DiffusionClient will behave. It is possible to create a new Connection object by

```
var connectionObject = new DiffusionClientConnectionDetails();
```

This will contain all of the defaults, another way of specifying connection details is to list the attributes that are not the defaults, as Diffusion™ Client will extend the supplied connection object with the defaults. As in the example above, all of the Connection Details will have their defaults set, apart from topic and the onDataFunction callback.

Events

The connection object has attributes that are listeners for certain events, if these are set in the connection object then they will be called when these events happen.

Event	**Condition**
onDataFunction	This is the function that will be responsible for handling mefrom Diffusion™. This function will be called with an argument of WebClientMessage. This function will be called even if there is a topic listener in place for a topic
onBeforeUnloadFunction	This function is called when the user closes the browser or navigates away from the page
onCallbackFunction	This function is called when Diffusion™ has connected, or exhausted all transports and cannot connect. This function is called with a boolean argument
onInvalidClientFunction	This function is called when an invalid Diffusion™ operation is called, for instance if Diffusion.subscribe was called before Diffusion.connect
onCascadeFunction	This function is called when the DiffusionClient cascades transports. The function is called with an argument of the {String} transport name, or NONE if all transport are exhausted
onPingFunction	This function is called with an argument of PingMessage when the ping response has been returned from the server
onAbortFunction	This function is called when the Diffusion™ Server has terminated the client connection (or the connection has been banned)
onLostConnectionFunction	This function is called when the DiffusionClient has lost connection with the Diffusion™ Server
onConnectionRejectFunction	This function is called when the DiffusionClient connection has been rejected by the Diffusion™ Server, this is due to incorrect credentials
onMessageNotAcknowledgedFunction	This function is called when a message that has been requested as Acknowledge did not respond in time

Copyright 2013 Push Technology

onServerRejectedCredentialsFunction	This function is called after a DiffusionClient.sendCredentials and the server rejected the credentials
onTopicStatusFunction	This function is called if the status of a subscribed topic changes (topic removal implemented for now)

Receiving messages

The onDataFunction with a class called WebClientMessage will only contain one message even if the messages sent from the Diffusion™ Server were batched. If this was the case then this method would get repeatedly called until all of the messages were exhausted. The WebClientMessage class wraps the message sent from the Diffusion™ Server with utility methods like isInitialTopicLoad() and getTopic. See the jsdoc for the full list of utility methods.

Sending messages

There are two ways of sending messages to the Diffusion™ Server, firstly is the *DiffusionClient.send(topic, message)* method. Secondly is to use the *sendTopicMessage* method. If user headers are required, then it is best to use the TopicMessage class. Below is an example of sending a message via the TopicMessage class.

```
var topicMessage = new TopicMessage("Echo", "This is a message");
topicMessage.addUserHeader("Header1");
topicMessage.addUserHeader("Header2");

DiffusionClient.sendTopicMessage(topicMessage);
```

Subscribing / Unsubscribing

To subscribe and unsubscribe use the DiffusionClient.subscribe(topic), DiffusionClient.unsubscribe(topic) methods respectively. The parameter can be a topic, a topic selector or a list of comma delimited topics

Topic Listeners

During the lifetime of the connection, it may be required to have modular components that are notified about topic messages, these are topic listeners. A topic listener will call a supplied function with a WebClientMessage object when the topic of the message matches the pattern supplied. It is also worth noting that the onDataMessage function will be called as well as the Topic Listener function. You can have many Topic Listeners on the same topic pattern if required. For example if you wanted to be notified about a particular topic then the following would be issued

```
DiffusionClient.addTopicListener(""^Logs$", onDataTradeEvent);
```

Note the syntax here, the `^ $` are *regular expression* characters, the above means that the listener is only interest in receive the message event if the topic is Logs. If the following was issued.

```
var listenerRef = DiffusionClient.addTopicListener("Logs", onLogEvent,
 this);
```

then any topic name that has Logs in it would match.

You will need to keep a hold of the reference if you wish to remove the topic listener at a later date.

Dependant files

`lib/DIFFUSION/diffusion.js` depends on a few other files in the same directory: `DiffusionClient.swf` and `DiffusionClient.xap`. These take part in the transport cascade and should not be removed.

Failover

The JavaScript client does not support autofailover. You can still implement this using the onLostConnectionFunction.

Copyright 2013 Push Technology

Service Topic Data

The JavaScript API provides a basic interface for using Service Topics.

The API consists of a Service Topic Handler to process responses and using the generic `DiffusionClient.command(..)` method to send service requests.

The common sequence to follow is:

1. Add a topic listener, to capture the service topic load message
2. Subscribe to the service topic
3. With the ITL from the service topic create a service topic handler
4. Remove the topic listener
5. Send command messages to the service
6. Process any response in the function passed to the handler

To create a handler using the `DiffusionClient.createService(TopicMessage, function)` you must pass in the ITL of the service topic and the function that will be called when a service response is recieved. This function will be called with a `CommandMessage` as an argument.

To make Service requests you must use the `DiffusionClient.command(string,string,TopicMessage)` method to send command messages. The first string is the command to send. The second string is a correlation ID for the response. The TopicMessage is the message sent to the client with the correct topic and any additonal headers or payload you want to send in the request.

The messages you send and process must conform to the *Service Topic Protocol*.

An ordinary topic listener must be used to get the ITL to create the service topic handler. This listener is not needed for any subsequent message processing and you are encouraged to remove it after you have the ITL.

The correlation ID should be unique and must be generated by you.

ActionScript API

The ActionScript API is bundled in a library called DiffusionTransport.swc. This can be embedded into a Flex / Flash or Air application. Full asdoc is issued with the product so the sections below provide a brief outline of the uses for the classes and examples of their use. The ActionScript Library is based on the Event Model. There are many different types of events that a DiffusionClient will dispatch. These will need to be registered before notification will happen.

Using the ActionScript API

DiffusionClient

DiffusionClient (com.pushtechnology.diffusion.DiffusionClient) is the main class that will be used. This class enables the user to set all of the connection and topic information.

Connection Example

```
// Get a new DiffusionClient
theClient = new DiffusionClient();

// Set up the transport mode
theClient.setTransportMode(DiffusionClient.CASCADE);

// Set everything to enable the cascading
theClient.setTopic("Trade");
theClient.setHost("192.168.52.2");
theClient.setPort(3095);
theClient.setURL("http://localhost:8080");
```

Copyright 2013 Push Technology

```
// Add the listeners
theClient.addEventListener(DiffusionConnectionEvent.CONNECTION,
onConnection);
theClient.addEventListener(DiffusionMessageEvent.MESSAGE, onMessages);
theClient.addEventListener(DiffusionTraceEvent.TRACE, onTrace);
theClient.addEventListener(DiffusionExceptionEvent.EXCEPTION, onException);
theClient.addEventListener(DiffusionPingEvent.PING, onPing);

// And away we go..
theClient.connect();
```

Setting credentials

If credentials are required by the Diffusion™ Server then use the *setCredentials* method on the *DiffusionClient* class. The *DiffusionClientCredentials* class takes a constructor argument of username and password. Please bear in mind, that these are only tokens and can contain any information that the *AuthorisationHandler* requires.

```
var credentials:DiffusionClientCredentials = new
DiffusionClientCredentials(username, password);
theClient.setCredentials(credentials);
```

Connection Event

The connection event contains information about the success of the connection attempt. Below is a coding example of the possibilities for the connect event.

```
public function onConnection(event:DiffusionConnectionEvent) : void {
 if(event.wasConnectionRejected()) {
  theClientIDBox.text = "Connection Rejected by Diffusion Server";
 }else if(event.wasClientAborted()) {
  theClientIDBox.text = "Connection aborted";
 } else if(event.isConnected()) {
  theClientIDBox.text = event.getClientID();
  theConnectedTransportLabel.text = theClient.getTransportMode();
 } else {
  theClientIDBox.text = "Connection failed " + event.getErrorMessage();
 }
}
```

It is possible to receive a connection event after you have successfully connected, which may be due to a lost connection, or in the case of *client aborted* the Diffusion™ Server has deliberately closed the client connection. This normally means that a publisher has aborted the connection and the client must not try and connect again.

onMessage Event

When messages arrive from the Diffusion™ Server on a subscribed topic, the *DiffusionMessageEvent* will be dispatched. Contained in the event is a TopicMessage object *TopicMessage* (com.pushtechnology.diffusion.TopicMessage). This class contains helper methods that surround the message itself, like getTopic() and isInitialTopicLoad. More information can be found in the asdoc for this API.

```
public function onMessages(event:DiffusionMessageEvent) : void {
 var message:TopicMessage = event.getTopicMessage();
 ....
```

Subscriptions

Once the client has connected, you will be able to issue subscribe and unsubscribe commands. The subscribe and unsubscribe methods take a string format, that may be a Topic selection pattern, a list of topics that are comma delimited or a single topic.

Copyright 2013 Push Technology

Send

Once connected it is possible to send messages to the Diffusion™ Server on a particular topic. To do this, use the send method.

```
theClient.send("Fred","Hello publisher that looks after Fred");
```

In the example above, the publisher that looks after topic Fred, will receive a `messageFromClient` notification. If a message with user headers or encoding is required, then you will need to use the *sendTopicMessage* method. A TopicMessage (com.pushtechnology.diffusion.TopicMessage) allows for the setting of user headers and message encoding

Ping

It is possible to ping the Diffusion™ Server. To receive the Ping response, the listener is added to the client.

```
theClient.addEventListener(DiffusionPingEvent.PING, onPing);
```

The resulting ping event has two attributes in it, firstly the time taken to do a round trip from the client to the Diffusion™ Server and back again. The second attribute is how many items are currently in the client queue at the server. This information will enable the client to get some vital connection information. It is down to the implementation of the client to specify the ping frequency, if at all required.

Topic Listeners

During the life time of the connection, it may be required to have modular components notified about topic messages - these are Topic Listeners. A Topic Listener will call a supplied function with a TopicMessage object when the topic of the message matches the pattern supplied. The Topic Listeners will be called in the order that they are added, and before the default DiffusionMessageEvent.MESSAGE, that will be called as well as the topic listener Event. You can have many Topic Listeners on the same topic pattern if required. The function supplied in charge of processing the message can signal that a message is 'consumed', returning TRUE. In this case this message will not be relayed to subsequent TopicListeners and the default listener. For example, if you wanted to be notified about a particular topic, then the following would be issued;

```
var listenerRef:String theClient.addTopicListener("^Logs$",
theLogsDataGrid.onMessage);
```

Note the syntax here, the ^ $ are regex pattern strings, the above means that the listener is only interest in receive the message event if the topic is Logs. If the following was issued.

```
var listenerRef:String theClient.addTopicListener("Logs",
theLogsDataGrid.onMessage);
```

then any topic name that has Logs in it would match. You will need to keep a hold of the reference if you wish to remove the topic listener at a later date.

Timed Topic Listeners

A Timed Topic Listener will call a supplied function with an array of TopicMessage objects when the topic of the message matches the pattern, and only if the time supplied by the arguments has expired. Otherwise the TopicMessage will be stored until the time expired. Note that the function in charge of processing the message cannot determine if a message is 'consumed' as you can do in a Topic Listener. For example if you wanted to be notified about a particular topic then the following would be issued;

```
var timedListenerRef:String theClient.addTimedTopicListener("^Logs$",
theLogsDataGrid.onMessage, 2000, false);
```

Copyright 2013 Push Technology

The third parameter is the frequency at which the function supplied will be called. The optional fourth parameter could be set if this function must be called, even if there are not any messages stored

Reconnection

The ActionScript API supports reconnection. If you have reconnection enabled if you lose you connection to a server to can reestablish it, using the same client ID and with the client subscriptions preserved. To reconnect you must use the `reconnect` method when you lose a connection. You cannot reconnect an aborted client.

```
var client:DiffusionClient = createClient();

function onConnectionEvent(event:DiffusionConnectionEvent) {
 if (event.hasLostConnection()) {
  client.reconnect();
 }
 else if (event.isConnected()) {
  if (event.isReconnected()) {
   // Succesful reconnection
  }
 }
}

function createClient():DiffusionClient {
 var client:DiffusionClient = new DiffusionClient();
 client.addEventListener(DiffusionConnectionEvent.CONNECTION,
 onConnectionEvent);
 return client;
}
```

The above code is an example of how to setup an event listener for connection events, if the connection has been lost how to reconnect and how to tell if you have successfully reconnected the client.

Failover

The ActionScript client supports autofailover. For more information you should refer to the *ActionScript Failover* documentation.

Silverlight API

The Silverlight API is bundled in an assembly called PushTechnology.Transports.dll. This can be embedded into a Silverlight application. Full ndoc documentation is issued with the product, so the sections below provide a brief outline of the uses for the classes and examples of their use. The Silverlight library is based on an asynchronous event model. There are a few events that the client object will invoke; these need to be subscribed to before notification can happen.

Using the Silverlight API

The DiffusionClient class is the main class that will be used. This class enables the user to set all of the connection and topic information.

Instantiation and connection example

```
 theClient = new DiffusionClient( Dispatcher );

   // Instantiate the server details object and the initial topic to
 subscribe to
   ServerDetails details = new ServerDetails( "http://localhost:8080",
 "SpotOnly" );
```

Copyright 2013 Push Technology

```
   // Add the server details to the client
   theClient.AddServerDetails( details );

            // Add the event listeners
            theClient.ConnectionStatus += DiffusionConnectionStatus;
            theClient.MessageReceived += theClient_MessageReceived;

            // Now connect
            theClient.Connect();
```

Setting credentials

If credentials are required by the Diffusion™ Server then use the `Credentials` property on the `DiffusionClient` class. The `DiffusionClientCredentials` class takes a constructor argument of `userName` and `password` . Please bear in mind that these are only tokens and can contain any information that Diffusion"s `AuthorisationHandler` requires.

```
theClient.Credentials = new DiffusionClientCredentials(
    "username", "password" );
```

Rejection of credentials event

If the credentials are rejected by the Diffusion™ server, a `ServerRejectedCredentials` event will be fired. This can be subscribed to thus:

```
 theClient.ServerRejectedCredentials +=
    ServerRejectedCredentials;
```

Message not acknowledged event

When a message is created with the "acknowledge" flag, this event will be fired when a message is not acknowledged by the Diffusion™ server within the specified time period. This can be subscribed to thus:

```
theClient.MessageNotAcknowledged += MessageNotAcknowledged;
```

The ConnectionStatus event

The `ConnectionStatus` event contains information about whether the connection was successful. Here follows an example of the usage of this event.

```
  /// <summary>
        /// Called when the connection state to Diffusion changes.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        void DiffusionConnectionStatus(
            object sender,
            DiffusionConnectionStatusEventArgs e )
        {
            switch( e.StatusType )
            {
                case DiffusionConnectionStatusType.ConnectionFailed:
                {
                    Dispatcher.BeginInvoke( () =>
                        MessageBox.Show( "Unable to connect to Diffusion.
 Diffusion reports: " + e.ExtraData,
                            "Connection failed", MessageBoxButton.OK ) );
                }
                break;

                case DiffusionConnectionStatusType.ConnectionReset:
                {
```

Copyright 2013 Push Technology

```
                    Dispatcher.BeginInvoke( () =>
                        MessageBox.Show( "The connection to Diffusion has
been reset.  Diffusion reports: " + e.ExtraData,
                            "Connection failed", MessageBoxButton.OK ) );
                }
                break;
            }
        }
```

Note that it is possible to receive a `ConnectionStatus` event after you have successfully connected; it may be due to a lost connection, or in the case of `ConnectionAborted` , the Diffusion™ server has closed the client connection. This normally means that a publisher has aborted the connection and the client must not try to connect again.

The MessageReceived event

When messages arrive from the Diffusion™ server on a subscribed topic, the `MessageReceived` event will be fired. This event contains a sender and a `TopicMessageEventArgs` object which itself contains a `TopicMessage` object which can be interrogated to discover the contents of the received message.

The TopicStatusMessageReceived event

When the status of a topic changes on the Diffusion™ server, the `TopicStatusMessageReceived` event will be fired. This event contains a sender and a `TopicStatusMessageEventArgs` object which contains the alias of the topic on which the status has changed. Currently, only the notification of the removal of a topic is implemented.

Subscriptions

Once the client has connected, you will then be able to issue `Subscribe` and `Unsubscribe` commands. These commands take string arguments which may be a Topic selection pattern , a list of topics that are comma-delimited, or a single topic.

Sending non-encoded topic messages

Once the client has connected, it is possible to send messages to the Diffusion™ server on a particular topic. To do this, use either the `Send` or `SendTopicMessage` methods on the DiffusionClient object, thus:

```
  theClient.Send( "Fred", "Hello, publisher that looks after Fred" );

            TopicMessage message =
                new TopicMessage( "Fred", "Hello, publisher that looks
after Fred" );

            theClient.SendTopicMessage( message );
```

Note that the `TopicMessage` itself contains methods to set (for instance) user headers and encoding, or the convenience methods described below can handle the alternate encoding scenarios.

In the above examples, the publisher that looks after the topic `Fred` will receive a `messageFromClient` notification internally.

Sending an encrypted topic message

Sending an encrypted topic message is achieved by calling the `SendTopicMessageEncrypted` method on the `DiffusionClient` object thus:

```
 theClient.SendTopicMessageEncrypted( new TopicMessage( "Fred", "Hello,
publisher that looks after Fred" ) );
```

This will set the relevant encoding flags on the message itself, and the message will be encrypted immediately prior to sending to the Diffusion™ server.

Copyright 2013 Push Technology

Note that because of the limitations of http-based transports, attempting to send a message of this type will result in a non-encoded message being sent.

Sending a compressed topic message

Sending a compressed topic message is achieved by calling the `SendTopicMessageCompressed` method on the `DiffusionClient` object thus:

```
  theClient.SendTopicMessageCompressed( new TopicMessage( "Fred", "Hello,
 publisher that looks after Fred" ) );
```

This will set the relevant encoding flags on the message itself, and the message will be compressed immediately prior to sending to the Diffusion™ server.

Note that because of the limitations of http-based transports, attemptingt to send a message of this type will result in a non-encoded message being sent.

Sending a Base64-encoded topic message

Sending a Base64-encoded topic message is achieved by calling the `SendTopicMessageBase64` method on the `DiffusionClient` object thus:

```
theClient.SendTopicMessageBase64( new TopicMessage( "Fred", "Hello,
 publisher that looks after Fred" ) );
```

This will set the relevant encoding flags on the message itself, and the message will be Base64-encoded immediately prior to sending to the Diffusion™ server.

Ping

It is possible to "ping" the Diffusion™ server. To process the ping response, the user monitors the `MessageReceived` event and checks for a message type of `PingServer` , thus:

```
   private void HandleServerPingMessage( TopicMessageEventArgs e )
{
 var message = e.Message as PingMessage;

 if( message != null )
 {
  tbElapsedTime.Text = message.ElapsedTime.ToString();
  tbQueueSize.Text = message.QueueSize.ToString();
 }
}
```

Fetch

Using the "Fetch" method, a client may send a request to the Diffusion™ server for the current state of a topic, which returns a state message to the client. A client may do this even if not subscribed to the topic.

Topic Listeners

During the lifetime of the connection, it may be required to have modular components that get notified about topic messages - these are known as "topic listeners". A topic listener will call a supplied function with a `TopicMessage` object when the topic of the message matches the pattern supplied. It is also worth noting that the `OnMessageReceived` event will be called as well as the topic listener event itself.

It is possible to have many topic listeners on the same topic pattern if required. For example, if you wanted to be notified about a particular topic then the following could be issued:

```
  instrumentListener = theClient.AddTopicListener(
```

Copyright 2013 Push Technology

```
                    "^SpotOnly$", ProcessInstruments, this );
```

Note the syntax here - the "^" and "$" are regular expression pattern strings; the above means that the listener is only interested in receiving the message if the topic is "SpotOnly".

Enabling JavaScript method invoking

If the user wishes to call JavaScript functions (and they are permitted to do so by the Silverlight runtime) then the following method call should be used:

```
    theClient.InitialiseJavaScriptMethodInvoking( HtmlPage.Window );
```

Listening to internal transport 'debug' messages

To subscribe to the internal log tracings of the Silverlight API, the user can subscribe to the `DiffusionTraceEvent` on the `DiffusionClient` object, thus:

```
theClient.DiffusionTraceEvent += theClient_DiffusionTraceEvent;
```

This will enable the user to monitor all internal debug messages within the Silverlight API.

iOS API

The file `lib/diffusioniostransport.zip` contains the static libraries and header files that comprise the Diffusion™ iOS API.

Full API documentation is issued with the product so the sections below provide a brief outline of the purpose of the classes and examples of their use. The iOS Library uses the Delegate Model. There are many different types of events that a DFClient object will dispatch. These will need to be registered before notification will take place

The API documentation is included also as an XCode docset. Once installed into XCode the Diffusion client for iOS can be browsed within the XCode Organizer/Documentation browser.

Copyright 2013 Push Technology

Figure 53: XCode documentation browser

DFClient

DFClient

The DFClient class is the main class that will be used. This class enables the user to set all of the connection and topic information.

Connection Example

```
NSURL *serverURL = [NSURL URLWithString:@"dpt://
demo.pushtechnology.com:80"];
DFServerDetails *server = [[[DFServerDetails alloc] initWithURL:serverURL ]
 autorelease];
DFConnectionDetails *cnxDetails = [[[DFConnectionDetails alloc]
 initWithServer:server topics:@"Echo" andCredentials:nil] autorelease];

dfClient = [[DFClient alloc] init];
dfClient.connectionDetails = cnxDetails;
dfClient.delegate = self; // Presumes that this class implements the
 DFClientDelegate protocol
[dfClient connect];
```

Diffusion Delegate

The Diffusion Delegate class is a custom class that must adhere to the DFClientDelegate protocol. The protocol consists of the following methods

```
/**
```

Copyright 2013 Push Technology

```
 Protocol implemented by classes wishing to receive notification from
 Diffusion.
 Notification primarily of new messages and the state of the connection to
 the server.
 */
@protocol DFClientDelegate

/**
 * This method will be called when the DFClient trys to connect, if the
 connection is made then isConnected will be true
 * @param isConnected
 */
- (void) onConnection:(BOOL) isConnected;

/**
 * This method will be called when the DFClient has lost connection to the
 Diffusion Server
 */
- (void) onLostConnection;

/**
 * This method will be called when the Diffusion Server has terminated the
 connection (barred)
 */
- (void) onAbort;

/**
 * This method will be called when a message has been received from the
 Diffusion Server.
 * This method will be called as well as any TopicListeners that might match
 the topic.
 */
- (void) onMessage:(DFTopicMessage *) message;

/**
 * This method will be called on receipt of the ping request
 * @see DFClient
 * @param message PingMessage
 */
- (void) onPing:(DFPingMessage *) message;

/**
 * This method will be called after a send credentials message, and the
 server rejected the credentials
 * @see DFClient
 */
- (void) onServerRejectedConnection;

/**
 * This method will be called if the server didn't respond to an Ack Message
 in time
 * @see TopicMessage
 */
- (void) onMessageNotAcknowledged:(DFTopicMessage *) message;

/**
 The list of DFServerDetails object has been exhausted, and no connection
 can be placed.
 Once this method is called the set of DFServerDetails is reset and further
 connections can be placed. In most simple scenarios where
 there is only one DFServerDetails object in the DFConnectionDetails object
 it should suffice to call method [client connect] here.
 @param client DFClient that has exhausted its set of DFServerDetails object
 from the DFClientDetails object.
```

Copyright 2013 Push Technology

```
 */
-(void)onConnectionSequenceExhausted:(DFClient*)client;

@optional

/**
 Conveys news from the Diffusion server that the named topic no longer
 exists
 */
-(void)onTopicRemoved:(NSString*) topicName;

/**
 The given DFServerDetails object has been selected for connection.
 @param details Details object that has been chosen.
 @param client DFClient that has chosen this DFServerDetails
 */
-(void)onConnectionDetailsAcquired:(DFServerDetails*)details forClient:
(DFClient*)client;
```

It is possible to receive an `onConnection` event after you have successfully connected, is may be due to a lost connection.

Credentials

When credentials are required, use the `credentials` property on the DFClient class. It is required that the user create a DFCredentials class and then set it on the client before calling connect.

onMessage Event

When messages arrive from the Diffusion™ Server on a subscribed topic, the `onMessage` method will be called on the delegate provided. The message is wrapped in a class called `TopicMessage` . This class contains helper methods that surround the message itself, such as the topic and isInitialLoad properties. More information can be found in the documentation for this API.

Subscriptions

Once the client has connected, you will be then able to issue subscribe and unsubscribe commands. The subscribe and unsubscribe methods take a string format, that may be a Topic selection pattern, a list of topics that are comma delimited or a single topic.

Send

Once connected it is possible to send messages to the Diffusion™ Server on a particular topic. To do this, use the send method.

```
[mClient send:"Fred" :"Hello publisher that looks after Fred"];
```

In the example above, the publisher that looks after topic Fred, will receive a `messageFromClient` notification. If sending a message with user headers then you will need to use the `sendTopicMessage` method. A TopicMessage allows for the setting of user headers

Ping

It is possible to ping the Diffusion™ Server. The delegate will be notified by the onPing method. The resulting ping event has two attributes in it, firstly the time stamp of the request. The second attribute is how many items are currently in the client queue at the server. This information will enable the client to get some vital connection information. It is down to the implementation of the client to specify the ping frequency, if at all required.

Topic Listeners

During the lifetime of the connection, it may be required to have modular components notified about topic messages - these are topic listeners. A topic listener will call a supplied method with a TopicMessage class when the topic of the message matches the topic name. Please note for performance the iOS and J2ME topic listeners do **not** have Regex

Copyright 2013 Push Technology

patterns but topic name matching. You can have many Topic Listeners on the same topic pattern if required. For example if you wanted to be notified about a particular topic then the following would be issued.

```
[mClient addTopicListener:aTopicListener];
```

Where aTopicListener implements the protocol `DFTopicListenerDelegate` which is shown below.

```
/**
 Protocol for receiving messages from a particular topic.
 */
@protocol DFTopicListenerDelegate

/**
 * This method is called if the TopicMessage matches the message received
 from Diffusion
 *
 * @param message
 * @return YES if the message is 'consumed' and should not be relayed to
 subsequent DFTopicListenerDelegate, nor the default listener.
 */
- (BOOL) onMessage:(DFTopicMessage *) message;

/**
 * This is the topic used to see if the message from Diffusion matches
 (equals) this String
 */
- (NSString *) getTopic;
```

Topic Listeners can be be removed by calling the removeTopicListener on the DFClient class

Installing the docset

Installing the Diffusion Client for iOS docset

1. Download the Diffusion Documentation Pack, from *http://download.pushtechnology.com/*
2. Unpack the ZIP file and locate the `iOS` folder.
3. With the iOS folder locate the file `com.pushtechnology.diffusion.ios.client.docset` and copy into one of the directories used by XCode for storing docsets (e.g. `~/Library/Developer/Shared/Documentation/DocSets`)

Copyright 2013 Push Technology

Alternatively, use the `Makefile` in the same directory and run `make install` to copy the same file into the same location.

4. Restart XCode if it's running

Android API

The Android API bundled in a library called diffusionandroidtransport.jar. Full API documentation is issued with the product so the sections below provides a brief outline of the uses for the classes and examples of their use. The DiffusionClient uses a listener model which closely resembles the external client API. The DiffusionConnectionListener interface is responsible for all of the event notifications from DiffusionClient

Using the Android API

DiffusionClient

DiffusionClient class is the main class that will be used. ConnectionDetails beside ServerDetails clasess enables the user to set all of the connection and topic information.

Connection Example

```
    // Get a new DiffusionClient
    theClient = new DiffusionClient();

    //Set the connection details
    ServerDetails serverDetails = new ServerDetials(new URL("dpt://
localhost:8080"));
    ConnectionDetails connectionDetails = new
 ConnectionDetails(serverDetails);
    theClient.setConnectionDetails(connectionDetails);

    // Make this listen to the DiffusionClient events
    theClient.setConnectionListener(this);

    // and let's go
    theClient.connect();
```

DiffusionConnectionListener

The DiffusionConnectionListener interface consists of the following methods (for further information refer to the Javadoc)

```
    /**
     * connected, will be called upon connection
     */
    void connected();

    /**
     * errorConnecting, will be called if there is an error connecting
     *
     * @param e
     */
    void errorConnecting(Exception e);

    /**
     * disconnected, called when the connection list lost
     */
    void disconnected();
```

Copyright 2013 Push Technology

```
    /**
     * connectionAborted, called when DiffusionServer has rejected the
connection
     */
    void connectionAborted();

    /**
     * onMessage, called when a message has been received from Diffusion
     *
     * @param message
     */
    void onMessage(Message message);

    /**
     * onPingMessage, called when a ping response is received
     *
     * @param message
     */
    void onPingMessage(PingMessage message);

    /**
     * onMessageNotAcknowledged, called when an ack message has not been
acknowledged by Diffusion
     *
     * @param message
     */
    public void onMessageNotAcknowledged(TopicMessage message);

    /**
     * onConnectionSequenceExhasted, called when the complete list of
ServerDetails have been exhausted.
     */
    public void onConnectionSequenceExhausted();

    /**
     * onConnectionDetailsAcquired, called each time a ServerDetails object
is selected for connection.
     *
     * @param serverDetails
     */
    public void onConnectionDetailsAcquired(ServerDetails serverDetails);

    /**
     * onServerRejectedCredentials, called when Diffusion reject the
credentials.
     */
    public void onServerRejectedCredentials() ;
```

Credentials

When credentials are required, there are three ways to set the credentials. The ServerDetails, the ConnectionDetails and the DiffusionClient all have a `setCredentials` method. It is required that the user create a DiffusionCredentials object and then pass to to one of these methods before calling connect. Only one of these ways should be used. If more than one way is used then the selection of the credentials to use is undefined. The Android API only supports sending credentials on connection.

onMessage Event

When messages arrive from the Diffusion™ Server on a subscribed topic, the `onMessage` will be called on the delegate provided. The message is wrapped in a interface called `Message` . This interface contains helper methods that surround the message itself, like getTopic() and isInitialTopicLoad. More information can be found in the documentation for this API.

Copyright 2013 Push Technology

Subscriptions

Once the client has connected, you will be then able to issue subscribe and unsubscribe commands. The subscribe and unsubscribe methods take a string format, that may be a Topic selection pattern, a list of topics that are comma delimited or a single topic.

Send

Once connected it is possible to send messages to the Diffusion™ Server on a particular topic. To do this, use the send method.

```
theClient.send("Fred","Hello publisher that looks after Fred");
```

In the example above, the publisher that looks after topic Fred, will receive a `messageFromClient` notification. If the message requires a user header you will need to use the *sendTopicMessage* method. A TopicMessage allows for the setting of user headers.

```
    TopicMessage message = new TopicMessage("Fred");
    message.addUserHeader("myHeaders");
    message.setMessage("Hello publisher that looks after Fred");
    theClient.sendTopicMessage(message);
```

Ping

It is possible to ping the Diffusion™ Server. The delegate will be notified of the response by the onPing function. The resulting ping event has two attributes in it, firstly the time stamp of the request. The second attribute is how many items are currently in the client queue at the server. This information will enable the client to get some vital connection information. It is down to the implementation of the client to specify the ping frequency, if at all required.

Topic Listeners

During the life time of the connection, it may be required to have modular components that get notified about topic messages, these are topic listeners. A topic listener will be called via its onMessage method with a Message object when the topic of the message matches the topic name. Please note for performance the iOS and JM2E topic listeners do **not** have Regex patterns but topic name matching. You can have many Topic Listeners on the same topic pattern if required. For example if you wanted to be notified about a particular topic then the following would be issued.

```
theClient.addTopicListener(topicListener);
```

Where topicListener implements the interface `TopicListener` which is shown below.

```
    /**
     * getTopic
     *
     * @return the topic that this listener is interested in. This will
     * not take REGEX expressions this needs to be an extact match
     */
    String getTopic();

    /**
     * onMessage
     *
     * @param message message which topic matches the getTopic method
     */
    void onMessage(Message message);
```

Topic Listeners can be be removed by calling the removeTopicListener on the DiffusionClient class

Threading Concerns

The Android DiffusionClient will create and dedicate a thread to listening to traffic from the Diffusion™ server and reacting to messages from it. Consequently methods on the DiffusionConnectionListener and

Copyright 2013 Push Technology

DiffusionTopicStatusListener will be executed in the same thread. Android does not allow background threads to interact with GUI controls, only the 'Main' thread is allowed to do so.

In order to overcome this, any non-main thread can pass a `java.lang.Runnable` to the 'Main' thread for execution via `android.view.View.post(Runnable action)` . For example ..

```
/*
 * Called when the Diffusion™ connection is established
 */
public void connected() {
    // Post this Runnable to the GUI thread to change the display
    String statusTextStr = String.format( "Connected to %s:%d\n", HOST,
 PORT );
    setStatus( statusTextStr );
    Log.i( TAG, statusTextStr );
  }

   /**
    * Set the content of the status view
    * @param statusStr
    */
   private void setStatus(final String statusStr)
    {
// Pass a Runnable to the GUI thread to execute via one of its widgets
     outputView.post( new Runnable() {
         public void run() {
            statusText.setText( statusStr );
        }
     } );
 }
```

User permissions in Manifest.xml

In oerder to establish a connection with the Diffusion™ Server, Android devices requeried to add the user permission *INTERNET* within the Manifest.xml. This permission allows applications to open network sockets.

J2ME API

The J2ME API bundled in a library called diffusionj2metransport.jar. Full API documentation is issued with the product so the sections below provides a brief outline of the uses for the classes and examples of their use. The DiffusionClient uses a listener model which closely resembles the external client API. The DiffusionConnectionListener interface is responsible for all of the event notifications from DiffusionClient

Using the DiffusionClient

Diffusion Client class is the main class that will be used. ConnectionDetails beside ServerDetails clasess enables the user to set all of the connection and topic information.

Connection Example

```
    // Get a new DiffusionClient
    theClient = new DiffusionClient();

    //Set the connection details
    ServerDetails serverDetails = new ServerDetials(new URL("dpt://
localhost:8080"));
    ConnectionDetails connectionDetails = new
 ConnectionDetails(serverDetails);
    theClient.setConnectionDetails(connectionDetails);
```

Copyright 2013 Push Technology

```
    // Make this listen to the DiffusionClient events
    theClient.setConnectionListener(this);

    // and let's go
    theClient.connect();
```

DiffusionConnectionListener

The DiffusionConnectionListener interface consists of the following methods

```
   /**
     * connected, will be called upon connection
     */
    void connected();

    /**
     * errorConnecting, will be called if there is an error connecting
     *
     * @param e
     */
    void errorConnecting(Exception e);

    /**
     * disconnected, called when the connection list lost
     */
    void disconnected();

    /**
     * connectionAborted, called when DiffusionServer has rejected the
connection
     */
    void connectionAborted();

    /**
     * onMessage, called when a message has been received from Diffusion
     *
     * @param message
     */
    void onMessage(Message message);

    /**
     * onPingMessage, called when a ping response is received
     *
     * @param message
     */
    void onPingMessage(PingMessage message);

    /**
     * onMessageNotAcknowledged, called when an ack message has not been
acknowledged by Diffusion
     *
     * @param message
     */
public void onMessageNotAcknowledged(TopicMessage message);

   /**
    * onConnectionSequenceExhasted, called when the complete list of
ServerDetails have been exhausted.
    */
public void onConnectionSequenceExhausted()
```

Copyright 2013 Push Technology

```
/**
    * onConnectionDetailsAcquired, called each time a ServerDetails object
is selected for connection.
    *
    * @param serverDetails
    */
public void onConnectionDetailsAcquired(ServerDetails serverDetails)

/**
    * onServerRejectedCredentials, called when Diffusion reject the
credentials.
    *
    * @param serverDetails
    */
public void onServerRejectedCredentials()
```

Credentials

When credentials are required, use the `setCredentials` method on the DiffusionClient class. It is required that the user create a DiffusionCredentials class and then set it on the client before calling connect.

onMessage Event

When messages arrive from the Diffusion™ Server on a subscribed topic, the `onMessage` will be called on the delegate provided. The message is wrapped in a interface called `Message` . This interface contains helper methods that surround the message itself, like getTopic() and isInitialTopicLoad. More information can be found in the documentation for this API.

Subscriptions

Once the client has connected, you will be then able to issue subscribe and unsubscribe commands. The subscribe and unsubscribe methods take a string format, that may be a Topic selection pattern, a list of topics that are comma delimited or a single topic.

Send

Once connected it is possible to send messages to the Diffusion™ Server on a particular topic. To do this, use the send method.

```
theClient.send("Fred","Hello publisher that looks after Fred");
```

In the example above, the publisher that looks after topic Fred, will receive a `messageFromClient` notification. If a message with user headers or then you will need to use the `sendTopicMessage` method. A TopicMessage allows for the setting of user headers

```
      TopicMessage message = new TopicMessage("Fred");
      message.addUserHeader("myHeaders");
      message.setMessage("Hello publisher that looks after Fred");
      theClient.sendTopicMessage(message);
```

Ping

It is possible to ping the Diffusion™ Server. The delegate will be notified by the onPing function. The resulting ping event has two attributes in it, firstly the time stamp of the request. The second attribute is how many items are currently in the client queue at the server. This information will enable the client to get some vital connection information. It is down to the implementation of the client to specify the ping frequency, if at all required.

Topic Listeners

Copyright 2013 Push Technology

During the life time of the connection, it may be required to have modular components that are notified about topic messages, these are topic listeners. A topic listener will call a suppled function with a Message object when the topic of the message matches the topic name. Please note for performance the iOS and JM2E topic listeners do **not** have Regex patterns but topic name matching. You can have many Topic Listeners on the same topic pattern if required. For example if you wanted to be notified about a particular topic then the following would be issued.

```
theClient.addTopicListener(topicListener);
```

Where topicListener implements the interface `TopicListener` which is shown below.

```
/**
 * getTopic
 *
 * @return the topic that this listener is interested in. This will
 * not take REGEX expressions this needs to be an extact match
 */
String getTopic();

/**
 * onMessage
 *
 * @param message message which topic matches the getTopic method
 */
void onMessage(Message message);
```

Topic Listeners can be be removed by calling the removeTopicListener on the DiffusionClient class

C API

The C API is provided as a source distribution which builds to either a dynamic or shared library on UNIX systems. It can then be linked into your own C/C++ applications, or used as the foundation for creating Diffusion™ clients for other languages which can be extended via binary APIs.

Using the C API

Building

To build the library, type `make` in the source directory. This should build shared and static versions of the Diffusion™ library, `libdiffusion.so` and `libdiffusion.a` respectively. Additionally, a number of sample applications should also be built.

Installation

The libraries should be copied to a location on the user's `LD_LIBRARY_PATH` (or equivalent). Two header files, `diffusion.h` and `llist.h` should be copied to your C compiler's include path.

For example, on Ubuntu 10.10 the following may be appropriate:

```
# mkdir /usr/local/include/diffusion
# cp diffusion.h llist.h /usr/local/include/diffusion
# mkdir /usr/local/lib/diffusion
# cp libdiffusion.* /usr/local/lib/diffusion
# echo "/usr/local/lib/diffusion" > /etc/ld.so.conf.d/libdiffusion.conf
# ldconfig
```

Copyright 2013 Push Technology

Example usage Connecting to a Diffusion™ instance (no credentials)

```
DIFFUSION_CONNECTION *c = diff_connect("localhost", 8080, NULL);
if(c == NULL) {
 fprintf(stderr, "Failed to connect to Diffusion\n");
 return(-1);
}
```

Connecting to a Diffusion™ instance (with credentials)

```
SECURITY_CREDENTIALS creds;
creds.username = strdup("smith");
creds.password = strdup("secret");
DIFFUSION_CONNECTION *c = diff_connect("localhost", 8080, &creds);
if(c == NULL) {
 fprintf(stderr, "Failed to connect to Diffusion\n");
 return(-1);
}
```

Requesting a subscription to a topic

```
DIFFUSION_CONNECTION *c = diff_connect( ... );

if(diff_subscribe(c, "Assets") == -1) {
 fprintf(stderr, "Failed to subscribe to topic\n");
 return(-1);
}
```

Using the event loop and callbacks

```
void on_initial_load(DIFFUSION_MESSAGE *msg) { ... }
void on_delta(DIFFUSION_MESSAGE *msg) { ... }

...

DIFFUSION_CONNECTION *c = diff_connect( ... );
diff_subscribe( ... );

DIFFUSION_CALLBACKS callbacks;
DIFF_CB_ZERO(callbacks);  // Reset callback structure
callbacks.on_initial_load = &on_inital_load;
callbacks.on_delta = &on_delta;

diff_loop(c, &callbacks);
```

DiffusionWrapper.js

The Diffusion™ wrapper is a script which addresses a weakness in the Flash and Silverlight VMs - when running inside a web browser there is no provision for the execution of callback code when the user closes either the containing tab or the entire browser window.

Copyright 2013 Push Technology

Consequently there is no opportunity for the Diffusion™ client to inform the server that the client is willingly closing the connection, instead the connection is severed. This can result in various server warnings (dependant on the transport mechanism i.e DPT or HTTP) as well as maintaining the server-side reference to the client if any 'keep-alive' properties are set.

Thankfully the Javascript environment in the hosting browser does provide "browser closing" callbacks, and these are employed by DiffusionWrapper.js. By supplying the `setClientDetails` function with client and server info after a connection has been established within the Flash or Silverlight API, DiffusionWrapper attaches a Javascript function to the `onbeforeunload` event that is triggered when a browser window or tab closes. This then notifies Diffusion™ that the client is deliberately closing.

`DiffusionWrapper.js` can be found in `html/lib/DIFFUSION/`. For reasons of backward compatibility It can also be found at `html/lib/DIFFUSION/FlashWrapper.js`

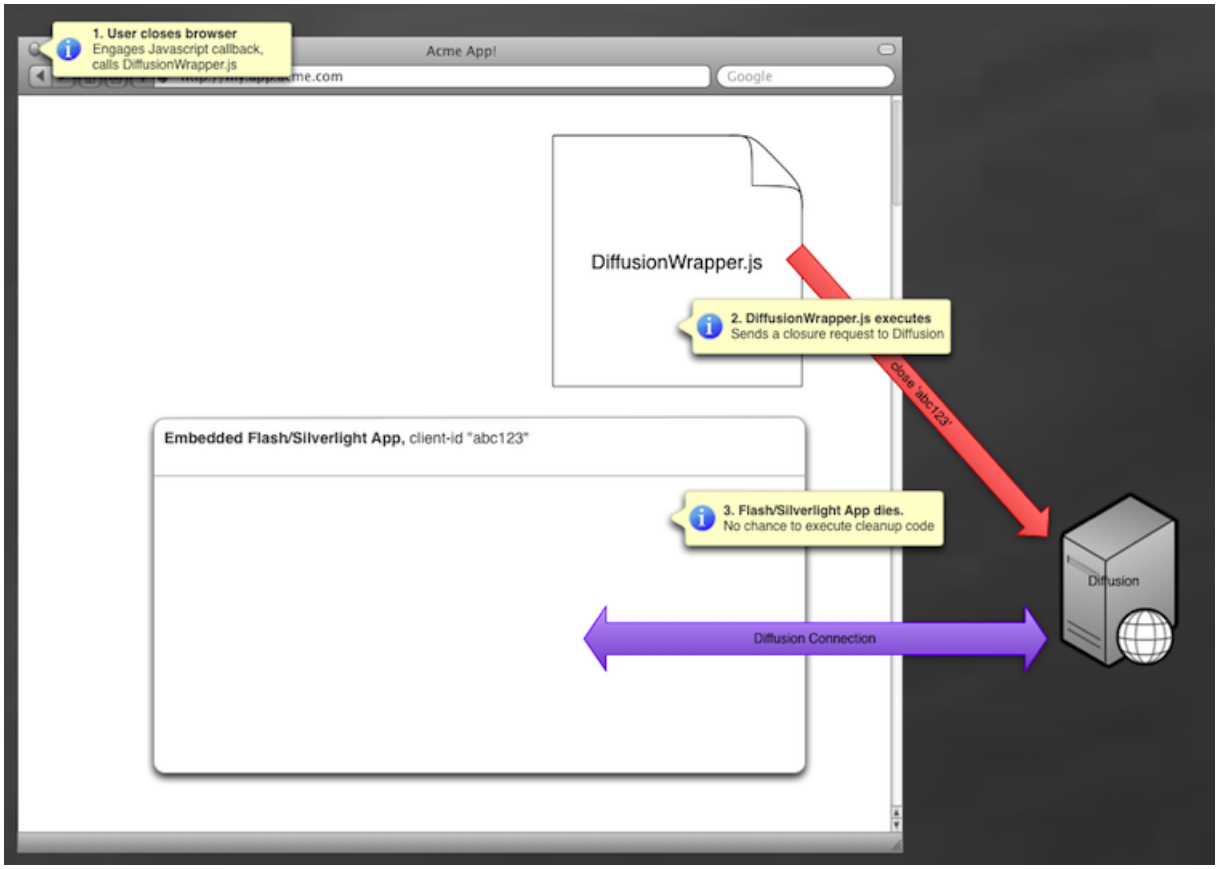


Figure 54: Wrapper

How to use DiffusionWrapper.js

The HTML page that loads the Flash/Silverlight app, has to load the DiffusionWrapper script.

```
<html>
 <head>
  <script src="DiffusionWrapper.js" language="javascript"></script>
 </head>
 <body>
 .....
 </body>
</html>
```

Copyright 2013 Push Technology

The Flash/Silverlight app makes a call to the external DiffusionWrapper method `setClientDetails`, typically in the `onConnection` callback, supplying the client ID and server Url. In the example below, the method `ExternalInterface.call("setClientDetails")` provides DiffusionWrapper with the client ID and server Url. DiffusionWrapper then adds a method to the `onbeforeunload` event of the window, which will inform the Diffusion Server that the client is intentionally closing the connection when the browser window or tab is closed.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                width="371" height="334" minWidth="955"
                minHeight="600" layout="absolute">
<mx:Script>
 import com.pushtechnology.diffusion.ConnectionDetails;
 import com.pushtechnology.diffusion.DiffusionClient;
 import com.pushtechnology.diffusion.ServerDetails;
 import com.pushtechnology.diffusion.events.DiffusionConnectionEvent;
 import com.pushtechnology.diffusion.events.DiffusionExceptionEvent;
 import com.pushtechnology.diffusion.events.DiffusionMessageEvent;


 private var theServerUrl:String = "http://127.0.0.1:8080"
 private var theInitialTopic:String = "Echo"
 private var theClient:DiffusionClient;

 private function onConnect():void{
     theClient = new DiffusionClient();
     var theConnectionDetails:ConnectionDetails = new ConnectionDetails(new
 ServerDetails(theServerUrl),null);

     // Add the listeners
     theClient.addEventListener(DiffusionConnectionEvent.CONNECTION,
 onConnection);
     theClient.addEventListener(DiffusionMessageEvent.MESSAGE, onMessages);
     theClient.addEventListener(DiffusionExceptionEvent.EXCEPTION,
 onException);

     //Lets Go...
     theClient.connect(theConnectionDetails);
 }

 private function onConnection(event:DiffusionConnectionEvent):void{
     // Is the client Connected?
     if(event.isConnected()){
         //Check the externalInterface avalibility
        if(ExternalInterface.available){
          // Send the ClientId and ServerURL to the DiffusionWrapper
          ExternalInterface.call("setClientDetails",
theClient.getClientID(), event.getServerDetails().getURL());
         }else{
         //The externalInterface is not available, so the DiffusionWrapper
 won't be called.
        }
   }
 }

 private function onMessages(event:DiffusionMessageEvent):void{
  //Message received
     theMessages.text += event.toString();
 }

 private function onException(event:DiffusionExceptionEvent):void{
```

Copyright 2013 Push Technology

```
  //Exception received
  theMessages.text += event.toString();
 }

</mx:Script>
 <mx:TextArea id="theMessages" x="18" y="40" width="335" height="283"
 text="Messages"/>
 <mx:Button id="bConnect" x="105.5" y="10" width="160" label="Connect"
 click="onConnect()"/>
</mx:Application>
```

Copyright 2013 Push Technology

Copyright 2013 Push Technology

Chapter 18

Configuration

Topics:

- *XML Configuration*
- *Programmatic Configuration*

How to configure Diffusion™

The Diffusion™ Server may be configured using XML files which would normally reside in the `etc` folder.

Alternatively, a Diffusion™ Server may be instantiated in a Java application and configured programatically. Some properties may also be changed at runtime programmatically (i.e from within Publishers).

In a Java Client environment certain properties may also be configured programatically.

All properties (whether configured from XML or programmatically) are available to read programmatically from within the Java API.

XML Configuration

Configuring a Diffusion™ Server using XML property files

XML Propery Files

A Diffusion™ Server is configured using a set of XML property files typically loaded from the `etc` folder. In a new Diffusion™ installation example versions of these files are provided which may be edited as required.

XML is used rather than standard property files due to the hierarchic nature and the ability to support repeating groups.

The Introspector has a built in configuration editor, which is able to load and save the configuration files remotely if required.

XSD files are issued that define the content of the XML property files and this section summarises the XSD content.

Configuration Path Loading

It is possible to pass a parameter to diffusion upon startup so that files are not automatically loaded from the `etc` folder but loaded from a different folder. This new folder doesn't need to contain the complete set of XML files, but the file will be loaded from the specified folder first, if it exists. If it doesn't then Diffusion™ will load the configuration file from the `etc` folder. When Diffusion™ starts, it will log where each configuration file has been loaded from.

Classpath Loading

When Diffusion™ starts, the `data` and `etc` folder are on the class path. The ext folder, and its sub-directories will be scanned for jar files and class loaded. This means that you can add new jars to the Diffusion™ runtime, without having to edit the startup scripts. Caution, needs to be taken then, when creating backup jars in the ext folder. Anything that ends in .jar will be class loaded.

XML Value Types

When XML values are loaded, the schema is checked so that we know that it is valid, but to aid configuration, there are some extra data types. When values are loaded, they are trimmed of leading and trailing white space.

Table 55: XML Value Types

Data Type	Meaning
push:boolean	true or false
push:string	String value
push:int	A number between -2,147,483,648 and 2,147,483,647
push:long	A number between -9,223,372,036,854,775,808 and 9,223,372,036,854,775,807
push:double	A number between 2-1074and (2-2-52)¬†.21023
push:port	A positive number but less than 65535
push:millis	A string that represents the number of Milliseconds. Append the Mnemonic for the time unit. The mnemonic can be either upper or lower case. **Table 56: XML Millis Mnemonics** **Mnemonic** / **Time Unit** s / Seconds

Copyright 2013 Push Technology

<table>
<tr><th>Data Type</th><th>Meaning</th></tr>
<tr><td></td><td>
<table>
<tr><th>Mnemonic</th><th>Time Unit</th></tr>
<tr><td>m</td><td>Minutes</td></tr>
<tr><td>h</td><td>Hours</td></tr>
<tr><td>d</td><td>Days</td></tr>
</table>
360000, 360s, 6m all represent 6 minutes
</td></tr>
<tr><td>push:bytes</td><td>A string that represents the number of bytes. Append the mnemonic size unit. The mnemonic can be either upper or lower case.
Table 57: XML Bytes Mnemonics
<table>
<tr><th>Mnemonic</th><th>Size Unit</th></tr>
<tr><td>k</td><td>Kilobytes</td></tr>
<tr><td>m</td><td>Megabytes</td></tr>
<tr><td>g</td><td>Gigabytes</td></tr>
</table>
6291456, 6144k, 6m, all represent 6 Megabytes
</td></tr>
<tr><td>push:log-level</td><td>A log level can be FINEST, FINE, INFO, ADVICE, WARNING or SEVERE</td></tr>
<tr><td>push:percent</td><td>A value that represents a percentage, this can have the trailing percent sign (%)</td></tr>
<tr><td>push:positiveNonZeroInt</td><td>A number between 1 and 2,147,483,647</td></tr>
<tr><td>push:positiveInt</td><td>A number between 0 and 2,147,483,647</td></tr>
<tr><td>push:threadPriority</td><td>A number between 1 and 10</td></tr>
<tr><td><element></td><td>This notation is used to indicate a complex element type. May also be List<element> to indicate a repeating property group.</td></tr>
</table>

Environmental Values

When defining custom configurations, it is possible to define environmental variables that can be reused in all XML property files. These variables can be defined in the *etc/Env.xml* property file in order to be used in all other property files. Suppose, for example, the *etc/Env.xml* file defines a `server-name` variable, with value `d-unit` as follows:

```
<env>
 <property name="server-name">d-unit</property>
</env>
```

The `server-name` variable can then be used in all other property files, where the value `d-unit` is appropriate, either as a value for an attribute, as in

```
<server name={server-name}>,Ä¶</server>
```

or as a name for an element as in:

```
<server>{server-name}</server>
```

Copyright 2013 Push Technology

As a side remark, it is worth noting that names can be combined to provide malleable environmental variables. Suppose for instance `Env.xml` contains the following entries:

```
<env>
 <property name="server-name">myServer</property>
 <property name ="server-version">V2.0</property>
</env>
```

Then `server-name` and `server-version` can be combined, for instance within the same *`etc/Env.xml`* , as

```
<property name="server-and-version">{server-name}-{server-version}</
property>
```

and used in all other configuration files.

Obfuscated Values

Obfuscation is a technique through which sensitive entries can be hidden in clear text. Within the Diffusion™ context, obfuscation can be used to hide password, and any other data deemed to be sensitive, to be included in configuration files. To create obfuscated values you can use the `propertyobfuscator` tool located in the `tools` directory and, once obfuscated, entries will be identified with a heading `OB` prefix in clear text.

Property Files

The remainder of this Topic defines the properties available in the major property files.

server

Server.xml - defines general Diffusion Server properties as well as multiplexers, security, conflation, client queues and thread pools.

server

All Server Properties

Name	Type	Description	Range
server-name	string	The server name is used to identify this server if running in a cluster. This will also be used as a prefix for client IDs. If not specified the local host name will be used.	[0 .. 1]
max-message-size	bytes	The maximum message size in bytes. This defines the maximum message size (including headers) that can be handled.	[1 .. 1]
default-load-message-capacity	bytes	The default capacity of a load message if not explicitly specified. If not supplied then 4096 is assumed.	[0 .. 1]
default-delta-message-capacity	bytes	The default capacity of a delta message if not explicitly specified. If	[0 .. 1]

Copyright 2013 Push Technology

Name	Type	Description	Range
		not specified then 1024 is assumed.	
message-length-size	int	Specifies the number of bytes utilised within Message headers to accommodate the Message length. This may have a value of 4, 2 or 1. This is a system wide setting which must match for all components of a distributed Diffusion system. Client APIs etc will automatically adapt to the size at the Server they connect to. The size chosen depends upon the maximum message size that needs to be catered for. A value of 1 will cater for Messages up to 127 bytes in length (including headers), 2 will cater for Messages up to 32,767 bytes in length and 4 can notionally cater for Messages up to 2,147,483,647 bytes on length. If not supplied then 4 is assumed.	[0 .. 1]
charset	string	The default character set to use for Diffusion message character conversions. See Java Encodings for full list. If this is not specified then "UTF-8" is assumed.	[0 .. 1]
multiplexers	*multiplexers*	Properties that define Multiplexers and how they are used.	[1 .. 1]
write-selectors	*write-selectors*	Properties that define Write Selectors and how they are used.	[0 .. 1]
security	*security*	Properties relating to security (optional).	[0 .. 1]
conflation	*conflation*	Conflation Policies and Topic to Policy mappings.	[0 .. 1]
client-queues	*client-queues*	Definitions of client queues.	[1 .. 1]
connection-timeouts	*connection-timeouts*	Timeout values relating to connections. If not	[0 .. 1]

Copyright 2013 Push Technology

Name	Type	Description	Range
		specified then defaults are assumed.	
date-formats	*date-formats*	Date and Time formats. If not specified then default formats will be assumed.	[0 .. 1]
thread-pools	*thread-pools*	Definitions of thread pools	[1 .. 1]
whois	*whois*	Definition of the WhoIs lookup service. If not specified then no WhoIs service will run.	[0 .. 1]
auto-deployment	*auto-deployment*	Automatic Deployment Properties (optional).	[0 .. 1]
geo-ip	*geo-ip*	Properties relating to the Geo IP lookup facility. If not specified then defaults will be assumed.	[0 .. 1]
usr-lib	*usr-lib*	User libraries (optional).	[0 .. 1]
hooks	*hooks*	User hooks used in the Server (optional)	[0 .. 1]

multiplexers

Properties that define Multiplexers and how they are used.

Name	Type	Description	Range
client	string	Name of the client multiplexer definition. If this is not specified then the first defined would be assumed.	[0 .. 1]
multiplexer-definition	*multiplexer-definition*	Multiplexer definition.	[1 .. -1]

multiplexer-definition

Multiplexer definition.

Attributes

Name	Type	Description	Required
name	string	The Multiplexer name.	true

Name	Type	Description	Range
size	positiveInt	This is the number of multiplexer instances that will start in readiness for clients to be assigned to. If there are going to be a large number of users,	[0 .. 1]

Copyright 2013 Push Technology

Name	Type	Description	Range
		then this number should be increased. If not specified then 2 is assumed.	
thread-priority	threadPriority	This is the thread priority that the multiplexer threads will run at. If not specified then 8 is assumed.	[0 .. 1]
load-balancer	string	This is the load balancer to use for for assigning connecting clients to multiplexer instances. There are currently two implemented load balancers, 'RoundRobin' and 'LeastClients'. RoundRobin is fast, although can not guarantee fairness of the connections per instance due to the randomness of disconnections. 'LeastClients' will guarantee fairness across the instances. If not specified then 'RoundRobin' is assumed.	[0 .. 1]
latency-warning	millis	Multiplexers are critical to the operation of Diffusion. If there are too many clients assigned to too few multiplexer instances then there is a possibility of message latency. This is an optional flag which can be set to issue a warning if the multiplexer instance is taking too long in its operational cycle (see ServerNotificationListener in the Publisher API). If this value is 0 then this feature will not be enabled. If not supplied then 0 is assumed.	[0 .. 1]
max-event-queue-size	positiveInt	This specifies the maximum size of the multiplexer event queue. This is the queue on which events from Publishers are queued for multiplexers and the default value should normally be more	[0 .. 1]

Copyright 2013 Push Technology

Name	Type	Description	Range
		than adequate. If this queue fills it can cause the Publisher threads to block until they can enqueue events and in this case it may be necessary to increase the value. This should normally be left at the default value which is 128k.	

write-selectors

Properties that define Write Selectors and how they are used.

Name	Type	Description	Range
thread-priority	threadPriority	The priority to run selector threads with. If not specified then Thread.NORM_PRIORITY is assumed.	[0 .. 1]
size	positiveNonZeroInt	Number of selector threads to run. If not specified then 1 is assumed.	[0 .. 1]
timeout	millis	Timeout for retry events in milliseconds. If not specified then 4000ms is assumed.	[0 .. 1]
load-balancer	string	This is the load balancer to use for for assigning connecting clients to selection instances. There are currently two implemented load balancers, 'roundrobin' and 'leastclients'. RoundRobin is fast, although can not guarantee fairness of the connections per instance due to the randomness of disconnections. 'leastclients' will guarantee fairness across the instances. If not specified then 'roundrobin' is assumed.	[0 .. 1]
queue-size	positiveNonZeroInt	Each selector thread is event driven. Events are first placed in a queue before being processed by the selector. The queue size	[0 .. 1]

Copyright 2013 Push Technology

Name	Type	Description	Range
		is configurable and defaults to 1024.	

security

Properties relating to security (optional).

Name	Type	Description	Range
authorisation-handler-class	string	This is the full name of a class, on the classpath, which implements the AuthorisationHandler interface in the Java Publisher API. If specified then the handler will be instantiated when the server starts and will be called to authorise client, connections, subscriptions and fetch requests.	[0 .. 1]

conflation

Conflation Policies and Topic to Policy mappings.

Name	Type	Description	Range
default-conflation-policy	string	The default Conflation policy. This specifies a Conflation Policy that will be used for any Topics that do not have explicit Conflation Policy mappings defined. If not specified then Conflation would not occur for Topics that do not have a Policy mapping defined. If specified it must be the name of a define Policy.	[0 .. 1]
conflation-policy	*conflation-policy*	Conflation Policy.	[0 .. -1]
topic-conflation	*topic-conflation*	Topic (or Topic pattern) to Policy mapping.	[0 .. -1]

conflation-policy

Conflation Policy.

Attributes

Name	Type	Description	Required
name	string	The Conflation Policy name.	true

Copyright 2013 Push Technology

Name	Type	Description	Range
mode	string	The conflation mode. This may have the value 'replace' or 'append'. If not specified then 'replace is assumed'. 'replace' means that when a matching message is found it is replaced by the new (or merged) message in its current queue position. 'append' means that when a matching message is found it is removed from its current queue position and the new (or merged) message is appended to the end of the queue. This option therefore preserves message ordering but there is the danger that messages are constantly sent to the end of the queue.	[0 .. 1]
matcher	string	The full class name of a message matcher of type com.pushtechnology.diffusion.api.conflation.MessageMatcher. If not supplied then a default matcher that matches by topic name will be used.	[0 .. 1]
merger	string	The full class name of a message merger of type com.pushtechnology.diffusion.api.conflation.MessageMerger. If not supplied then no merging with the new message will occur but the new message will either replace the existing one or the existing one will be removed and the new one appended to the end of the queue depending upon the mode.	[0 .. 1]

topic-conflation

Topic (or Topic pattern) to Policy mapping.

Name	Type	Description	Range
topic	string	The name of a topic or a topic selector pattern that indicates the topic(s) that	[1 .. 1]

Copyright 2013 Push Technology

Name	Type	Description	Range
		the specified Conflation Policy should be applied to.	
policy	string	The name of a configured Conflation Policy that should be applied to the specified topic(s)	[1 .. 1]

client-queues

Definitions of client queues.

Name	Type	Description	Range
default-topic-message-comparator-class	string	NOT USED - remains for backwards compatibility only	[0 .. 1]
default-queue-definition	string	The name of the queue definition to use by default. Connectors that do not explicitly specify a queue definition will use the one specified here.	[1 .. 1]
queue-definition	*queue-definition*	Queue Definition.	[1 .. -1]

queue-definition

Queue Definition.

Attributes

Name	Type	Description	Required
name	string	The Queue Definition name.	true

Name	Type	Description	Range
max-depth	positiveInt	The maximum depth of the queue. If the number of messages queued for a Client exceeds this number then the Client will be disconnected.	[1 .. 1]
conflates	boolean	Specifies whether Conflation should be applied to all Clients using this queue definition. If not specified then Conflation would not be applied by default.	[0 .. 1]
upper-threshold	percent	This specifies a percentage of the maximum queue size and if this value is	[0 .. 1]

Copyright 2013 Push Technology

Name	Type	Description	Range
		reached then any listeners (see ClientListener in the Publisher API) will be notified. Notification will occur only once and will not occur again until the queue has returned to the lower threshold. If not specified then no upper limit notification will occur.	
lower-threshold	percent	This specifies a percentage of the maximum queue size and indicates the level at which listeners (see ClientListener in the Publisher API) will be notified after an upper limit notification has occurred and the queue size has dropped back to the specified lower limit. If not specified then no lower limit notification will occur.	[0 .. 1]
auto-fragment	boolean	If a message is too large to fit into the output buffer and this option is set to true, the message will be automatically fragmented to 80% of the output buffer size. As this will happen per client, it may be highly inefficient. Consider creating your messages inside your publisher with fragmentation options instead. If not specified then false is assumed.	[0 .. 1]

connection-timeouts

Timeout values relating to connections. If not specified then defaults are assumed.

Name	Type	Description	Range
write-timeout	millis	This is the time in milliseconds that will be allotted to performing a single write to a Client. If the write has not completed within this limit then the Client will be disconnected. If	[0 .. 1]

Copyright 2013 Push Technology

Name	Type	Description	Range
		not specified then 2s is assumed.	
connection-timeout	millis	This is the time in milliseconds allowed for a connection to take place and complete its handshake processing. If not specified then 2s is assumed.	[0 .. 1]

date-formats

Date and Time formats. If not specified then default formats will be assumed.

Name	Type	Description	Range
date	string	T The format used when displaying dates. This should be specified according to the Java SimpleDateFormat specification. If not specified then "yyyy-MM-dd" is assumed.	[0 .. 1]
time	string	The format used when displaying times. This should be specified according to the Java SimpleDateFormat specification. If not specified then "HH:mm:ss" is assumed.	[0 .. 1]
date-time	string	The format used when displaying date and time. This should be specified according to the Java SimpleDateFormat specification. If not specified then "yyyy-MM-dd HH:mm:ss" is assumed.	[0 .. 1]
timestamp	string	The format used when displaying a timestamp, for example in a log, to millisecond precision. This should be specified according to the Java SimpleDateFormat specification. If not specified then "yyyy-MM-dd HH:mm:ss.SSS" is assumed.	[0 .. 1]

Copyright 2013 Push Technology

thread-pools

Definitions of thread pools

Name	Type	Description	Range
inbound	string	Name of the inbound thread pool definition.	[1 .. 1]
outbound	string	This property is deprecated and should no longer be used.	[0 .. 1]
background-thread-size	int	Number of threads to use for the background thread pool. If not specified then 10 is assumed.	[0 .. 1]
writer-selectors	positiveInt	This property is deprecated and should no longer be used. See new writer-selectors element for outbound selector configuration.	[0 .. 1]
thread-pool-definition	*thread-pool-definition*	Thread Pool Definition.	[1 .. -1]

thread-pool-definition

Thread Pool Definition.

Attributes

Name	Type	Description	Required
name	string	Name of the thread pool definition.	true

Name	Type	Description	Range
core-size	positiveInt	The core number of threads to have running in the thread pool. Whenever a thread is required a new one will be created until this number is reached, even if there are idle threads already in the pool.	[1 .. 1]
max-size	positiveInt	The maximum number of threads that may be created in the thread pool before tasks are queued. Such threads are released immediately after execution.	[1 .. 1]
queue-size	positiveInt	The Thread Pool queue size. When the max-size is reached then tasks will	[1 .. 1]

Copyright 2013 Push Technology

Name	Type	Description	Range
		be queued. If the value is zero then the queue is unbounded. If not 0 then the value must be at least 10.	
keep-alive	millis	The time to keep inactive threads alive for. This does not apply to core threads. If this is no specified then 0 is assumed.	[0 .. 1]
priority	threadPriority	This is the priority at which the threads will run. If not specified then 5 is assumed.	[0 .. 1]
thread-pool-listener	*thread-pool-listener*	Thread Pool Listener details (optional)	[0 .. 1]
rejection-handler-class	string	The name of a class implementing the ThreadPoolRejectionHandler interface which will be called if a task cannot be executed by the Thread Pool. If not specified then a default rejection policy is used so that rejected tasks are executed in the calling thread. The default rejection policy is implemented by the class com.pushtechnology.diffusion.api.threads.ThreadService.CallerRunsRej A thread will be rejected if all the threads are in use and the queue is full.	[0 .. 1]

thread-pool-listener

Thread Pool Listener details (optional)

Name	Type	Description	Range
queue-notification-handler-class	string	The name of a class implementing the ThreadPoolNotificationHandler interface which will be instantiated to handle notifications on the thread pool.	[1 .. 1]
queue-upper-threshold	percent	The size of the thread pool queue at which the notification handler will be called on the queueUpperThresholdReached	[1 .. 1]

Copyright 2013 Push Technology

Name	Type	Description	Range
		method. The method will be called once only until the queue size drops below the specified lower threshold.	
queue-lower-threshold	percent	The size of the thread pool queue at which the notification handler will be called on the queueLowerThresholdReached method if the upper threshold has previously been breached.	[1 .. 1]

whois

Definition of the WhoIs lookup service. If not specified then no WhoIs service will run.

Name	Type	Description	Range
provider	string	Name of the WhoIs provider class that must be on the classpath and must implement the API class WhoIsProvider. If not specified then WhoIsDefaultProvider is assumed.	[0 .. 1]
threads	int	The number of background threads that will process WhoIs resolver requests. If not specified then 2 is assumed. If set to 0 the service will not be started.	[0 .. 1]
host	string	The host name of a WhoIs provider which adheres to the RFC3912 WhoIs protocol. If not specified then "whois.ripe.net" is assumed.	[0 .. 1]
port	port	The port number that the WhoIs provider listens on. If not specified then 43 is assumed which is the normal value.	[0 .. 1]
whois-cache	*whois-cache*	Details of the WhoIs service cache which is used to cache WhoIs lookup results. If not specified then default values are assumed.	[0 .. 1]

Copyright 2013 Push Technology

whois-cache

Details of the WhoIs service cache which is used to cache WhoIs lookup results. If not specified then default values are assumed.

Name	Type	Description	Range
maximum	int	The maximum size of the WhoIs cache. When the cache size exceeds this number it will be tidied. A value of 0 means the cache will grow indefinitely unless entries are removed because they have exceeded their retention time. If not specified then 1000 is assumed.	[0 .. 1]
retention	millis	The time for which WhoIs cache entries are retained before being deleted. A value of 0 means entries are retained indefinitely or until the cache reaches its maximum size. If not specified then 0 is assumed.	[0 .. 1]
tidy-interval	millis	The interval at which the Whois cache tidier task will check if any cache entries have passed their retention time or if the cache has exceeded its maximum size. This is ignored if both maximum and retention are 0. If not specified then 1 minute is assumed.	[0 .. 1]

auto-deployment

Automatic Deployment Properties (optional).

Name	Type	Description	Range
directory	string	The name of the automatic deployment directory.	[1 .. 1]
scan-frequency	millis	The frequency at which the deployment directory is scanned for new deployments. If this is not specified then 5 seconds is assumed.	[0 .. 1]

Copyright 2013 Push Technology

geo-ip

Properties relating to the Geo IP lookup facility. If not specified then defaults will be assumed.

Attributes

Name	Type	Description	Required
enabled	boolean	Set to true to enable Geo IP lookup. This needs to be set to true if you are going to use connection or subscription validation policies. If not specified then true is assumed.	false

Name	Type	Description	Range
file-name	string	The name of the Maxmind GeoCityIP city file. If not specified then "../data/GeoLiteCity.dat" is assumed.	[0 .. 1]

usr-lib

User libraries (optional).

Name	Type	Description	Range
directory	string	Directory to load classes from. When the server starts, this folder is traversed, including sub directories and all jars or zip files added to the class loader.	[1 .. -1]

hooks

User hooks used in the Server (optional)

Name	Type	Description	Range
startup-hook	string	This is the class name of a class that implements the interface com.pushtechnology.diffusion.api.publisher.ServerStartupHook. If specified then the hook will be instantiated and the serverStarting method called when the server is starting, before the loading of publishers.	[0 .. 1]
shutdown-hook	string	This is the class name of a class that implements the interface com.pushtechnology.diffusion.api.publisher.ServerShutdownHook.	[0 .. 1]

Copyright 2013 Push Technology

Name	Type	Description	Range
		If specified then the hook will be instantiated and the serverStopping method called when the server is stopping.	

connectors

Connectors.xml - defines the Connectors required.

connectors

Connectors

connector

Connector Definition

Attributes

Name	Type	Description	Required
name	string	The Connector name	true

Name	Type	Description	Range
type	string	The type of connection supported. By default 'all' types are supported but the connector can be restricted to one of the following specific types - 'client' (Clients only), 'event' (Event Publishers only), or 'policy' (Policy File Requests only).	[0 .. 1]
host	string	The name or the IP Address that the connector will bind to. This is optional.	[0 .. 1]
port	port	The port on which the connector will accept connections.	[1 .. 1]
acceptors	positiveNonZeroInt	The number of acceptors to run for this connector. If this is not specified then 2 is assumed.	[0 .. 1]
backlog	positiveNonZeroInt	The maximum queue length for incoming clients. If a connection indication arrives when the queue is full, the connection is refused. If not specified then 1000 is assumed.	[0 .. 1]

Copyright 2013 Push Technology

Name	Type	Description	Range
socket-conditioning	*socket-conditioning*	Describes the properties associated with TCP socket connections.	[1 .. 1]
web-server	string	If this connector is required to serve HTTP requests then this should specify a web-server entry in WebServer.xml. If not specified then the Connector will not be able to serve HTTP requests.	[0 .. 1]
policy-file	string	The location/name of the policy file, if this connector is required to act as a policy file server (type='all' or 'policy').	[0 .. 1]
validation-policy-file	string	The location/name of a connection validation policy file to use for this connector. Applies only to type 'all' or 'client'.	[0 .. 1]
key-store	*key-store*	Key store details for any connector that is to support secure (SSL) connections. If not specified then SSL connections will not be supported.	[0 .. 1]
queue-definition	string	Optional queue definition to use for this connector. This only applies to connectors of type 'all' or 'client'. The definition must exist in Server.xml. If not specified then the default queue definition specified in Server.xml will be used.	[0 .. 1]
reconnect	*reconnect*	Optional reconnection properties which apply only to connectors that accept 'client' connections. If not specified then reconnection of client connections would not be supported.	[0 .. 1]
ignore-errors-from	*ignore-errors-from*	Specifies addresses from which connection errors should be ignored. This is useful for masking errors that might be reported due to the connector port being	[0 .. 1]

Copyright 2013 Push Technology

Name	Type	Description	Range
		pinged by some known external entity.	
thread-pool-definition	string	Optionally may be used to specify a thread pool definition to be used for this Connector to create its own inbound thread pool. If specified then the definition must exist in Server.xml. If not specified then the default inbound thread pool would be used.	[0 .. 1]
system-ping-frequency	millis	This indicates the frequency at which Clients should be pinged by the server to ensure that they are still connected. The time is the amount of time since the last activity (inbound message) from the client. If the time expires then a ping message will automatically be sent to the client. If a response is not received from the client before the expiry of another interval period then the Client is assumed to be disconnected. If this is not specified (or a value of 0 is supplied) then Clients will not be automatically pinged.	[0 .. 1]
fetch-policy	*fetch-policy*	Specifies a policy for batching fetch requests. If not specified then no policy will be applied (i.e. fetches will not be batched).	[0 .. 1]

socket-conditioning

Describes the properties associated with TCP socket connections.

Name	Type	Description	Range
input-buffer-size	bytes	Specifies the size of the socket input buffer to use for each connection. This must be large enough to accomodate the largest inbound message expected.	[0 .. 1]

Copyright 2013 Push Technology

Name	Type	Description	Range
		If not specified then 4k is assumed.	
output-buffer-size	bytes	This value specifies the size of the output buffer to use for each connection. This must be large enough to accomodate the largest message to be sent. Messages are 'batched' into this buffer and so the larger the buffer, the more messages can be sent in a single write. If this is not specified then 64k is assumed.	[0 .. 1]
keep-alive	boolean	This enables or disables TCP Keep alive. If not specified then true is assumed.	[0 .. 1]
no-delay	boolean	This enables or disables TCP_NODELAY (disable/enable Nagle's algorithm). If not specified then true is assumed.	[0 .. 1]
reuse-address	boolean	When a TCP connection is closed the connection may remain in a timeout state for a period of time after the connection is closed (typically known as the TIME_WAIT state or 2MSL wait state). For applications using a well known socket address or port it may not be possible to bind a socket to the required SocketAddress if there is a connection in the timeout state involving the socket address or port. Enabling this feature allows the socket to be bound even though a previous connection is in a timeout state. If not specified then the feature is enabled.	[0 .. 1]

key-store

Key store details for any connector that is to support secure (SSL) connections. If not specified then SSL connections will not be supported.

Copyright 2013 Push Technology

Attributes

Name	Type	Description	Required
mandatory	boolean	If true then all connections must use this keystore (i.e. SSL connection is mandatory). If not specified then false is assumed, meaning that the Connector would accept either SSL or non SSL connections.	false

Name	Type	Description	Range
file	string	The keystore file path.	[1 .. 1]
password	string	The password for the keystore.	[1 .. 1]

reconnect

Optional reconnection properties which apply only to connectors that accept 'client' connections. If not specified then reconnection of client connections would not be supported.

Name	Type	Description	Range
keep-alive	millis	This specifies the period within which a disconnected client can reconnect to the same Client session. Messages for the Client will continue to be queued during this period.	[1 .. 1]
max-depth	positiveInt	As messages will continue to be queued for a Client whilst it is disconnected this allows a larger maximum queue size to be specified which will be used during the period that the Client is disconnected. When the Client reconnects the maximum would revert back to its previous size (once any backlog had been cleared). If the specified size is not greater than the current maximum size then this would have no effect. If this is not specified then 0 is assumed which would mean that no attempt is made to extend	[0 .. 1]

Copyright 2013 Push Technology

Name	Type	Description	Range
		the queue size when a Client is disconnected.	

ignore-errors-from

Specifies addresses from which connection errors should be ignored. This is useful for masking errors that might be reported due to the connector port being pinged by some known external entity.

Name	Type	Description	Range
ip-address	string	An IP Address or unknown if the remote IP Address is being masked	[1 .. -1]

fetch-policy

Specifies a policy for batching fetch requests. If not specified then no policy will be applied (i.e. fetches will not be batched).

Name	Type	Description	Range
batch-size	positiveInt	Specifies the maximum number of fetch reply messages to send per batch. If this is set to 0 then no batching will occur.	[1 .. 1]
delay	millis	Specifies the time period between submissions of batches. If a batch size is specified then this must be a positive value.	[1 .. 1]

publishers

Publishers.xml - defines Publishers.

publishers

The set of publishers that the Diffusion server is aware of at startup.

publisher

A Publisher Definition.

Attributes

Name	Type	Description	Required
name	string	The Publisher name.	true

Name	Type	Description	Range
topics	string	An optional comma separated list of topic names specifying Topics to be automatically created	[0 .. 1]

Copyright 2013 Push Technology

Name	Type	Description	Range
		for the Publisher as it is started. This technique does not allow for Topics to be set up with data and so it is more usual to define the Topics you require in the initialLoad method of the Publisher. This property remains mostly for backwards compatibility.	
class	string	The full class name of a Java class that implements the Publisher. This class must extend the Java API Publisher class and provide implementations of methods as required. The class file must be available on the classpath of the Diffusion Server (or in the configured usr-lib or ext folder).	[1 .. 1]
enabled	boolean	By default the Publisher will be loaded as the Server starts but by setting this to false, the publisher will not be loaded.	[0 .. 1]
start	boolean	By default the Publisher will be started after it is loaded. By specifying this as false the Publisher can be loaded but not started - and then could be started later using JMX.	[0 .. 1]
topic-aliasing	boolean	Specifies whether Topic Aliasing should be turned on for all Topics created by the Publisher. If true then a short Topic alias will be transmitted in delta Messages instead of the full Topic name. By default this is true but because there are certain limitations when using Topic Aliasing there might be situations where you would want to turn it off.	[0 .. 1]
ack-timeout	millis	This specifies the default ACK (message acknowledgement) timeout	[0 .. 1]

Copyright 2013 Push Technology

Name	Type	Description	Range
		value (in milliseconds) to use for messages sent from the Publisher that require acknowledgement and do not have a timeout explicitly specified. If not specified then 1s is assumed.	
auto-ack	boolean	Indicates whether Messages sent from Clients to the Publisher requiring acknowledgement should be automatically acknowledged. By default this is false so Messages requiring acknowledgement would need to be manually acknowledged by the Publisher.	[0 .. 1]
subscription-policy-file	string	Path of a Subscription Validation Policy File. If specified then the file will be used to validate Client subscriptions to Topics owned by the Publisher.	[0 .. 1]
stop-server-if-not-loaded	boolean	If this is set to true and the publisher fails to load, then the Diffusion Server will be stopped. By default this is false.	[0 .. 1]
log-level	log-level	T Specifies the log level for the publisher. If not specified then the Publisher will log at the default log level.	[0 .. 1]
server	*server*	T A specification of a Server that will automatically be connected to by the Publisher when it starts.	[0 .. -1]
web-server	*web-server*	If the publisher has associated web content, it can be deployed with the publisher by specifying this property.	[0 .. 1]
launch	*launch*	Launch detail : Describes how the publisher might be accessed externally, if it has an associated webpage.	[0 .. -1]

Copyright 2013 Push Technology

Name	Type	Description	Range
property	*property*	A Property available to the Publisher. This can be used to configure Publisher specific variables or parameters.	[0 .. -1]

server

T A specification of a Server that will automatically be connected to by the Publisher when it starts.

Attributes

Name	Type	Description	Required
name	string	Server definition name.	true

Name	Type	Description	Range
host	string	The host name or IP address of the server.	[1 .. 1]
port	port	The port number that the server is listening on for Publisher Client connections from other Publishers.	[1 .. 1]
ssl	boolean	If true then the connection to the server will be a secure connection over SSL. In this case the specified port must represent an SSL client connector at the server. The keystore properties must also be supplied for secure connections. By default this is false.	[0 .. 1]
keystore-file-location	string	The path of the KeyStore file defining the SSL context. This is ignored if ssl=false but mandatory if it is true.	[0 .. 1]
keystore-password	string	The Keystore password. This is ignored if ssl=false and required if it is true.	[0 .. 1]
input-buffer-size	bytes	Specifies the size of the input buffer to use for the connection with the server. This is used to receive messages from the server and should ideally be the same size as the output buffer used at the server.	[1 .. 1]

Copyright 2013 Push Technology

Name	Type	Description	Range
output-buffer-size	bytes	The size of the output buffer to use for the connection with the server. This is used to send messages to the Server and should ideally match the size of the input buffer used by the server.	[1 .. 1]
fail-policy	string	This specifies what should happen if the Publisher fails to connect to the server. 'default' means that if unable to connect then no action will be taken and it would be the publisher's responsibility to handle this. 'close' means that if unable to connect to the server then the publisher will close. 'retry' means that if unable to connect then the connection will be automatically retried at intervals as specified by the retry-interval property.	[1 .. 1]
retry-interval	millis	If the fail-policy for a server is 'retry' then this is the interval at which the connection to the server will be retried. If not specified then 5s is assumed.	[0 .. 1]
credentials	*credentials*	Credentials to use for the server connection. If not specified then no credentials are passed on connection.	[0 .. 1]
queue-definition	string	Optional outbound queue definition for this server connection. The definition must exist in Server.xml. This defines the queue to use for outbound messages from the Publisher to the Server. If not specified then the default queue defintion in Server.xml will be used.	[0 .. 1]

credentials

Credentials to use for the server connection. If not specified then no credentials are passed on connection.

Copyright 2013 Push Technology

Name	Type	Description	Range
username	string	User name.	[0 .. 1]
password	string	Password.	[0 .. 1]

web-server

If the publisher has associated web content, it can be deployed with the publisher by specifying this property.

Name	Type	Description	Range
virtual-host	string	The name of the virtual host to deploy to (if not supplied, default-files-default is used).	[0 .. 1]
alias-file	string	T he alias file to use for this publisher	[1 .. 1]

launch

Launch detail : Describes how the publisher might be accessed externally, if it has an associated webpage.

Attributes

Name	Type	Description	Required
name	string	The Launcher name.	true
category	string	An optional category to which this Launcher belongs (e.g. "demo" for the Diffusion demo landing page)	false

Name	Type	Description	Range
description	string	A short description of this launcher.	[0 .. 1]
url	string	The URL at which a webpage associated with this Publisher can be found.	[1 .. 1]
icon	string	A URL or path at which an icon representing this launcher can be reached.	[0 .. 1]

property

A Property available to the Publisher. This can be used to configure Publisher specific variables or parameters.

Attributes

Name	Type	Description	Required
name	string	The Property Value	true

Copyright 2013 Push Technology

Name	Type	Description	Required
type	string	An optional property type. Usage of this is implementation specific.	false

web-servers

WebServer.xml - definitions of one or more Web Servers.

web-servers

Definitions of one or more Web Servers.

web-server

Web Server Definition.

Attributes

Name	Type	Description	Required
name	string	Name of the Web Server definition.	true

client-service

Optional Client Service.

Attributes

Name	Type	Description	Required
name	string	Client Service Name.	true
debug	boolean	Set true to debug the Client service. If not specified then false.	false

Name	Type	Description	Range
message-sequence-timeout	millis	This is used with HTTP clients to indicate how long to wait for a missing message is a sequence of messages before assuming it has been lost and closing the Client connection. If not specified then 2s is assumed.	[0 .. 1]
websocket-origin	string	This is used to control access from client web socket to Diffusion. This is a REGEX pattern that will match the origin of the request. A value of ".*" matches anything so all requests would be allowed. If this is not specified then	[0 .. 1]

Copyright 2013 Push Technology

Name	Type	Description	Range
		the service will be unable to handle Web Socket requests.	
cors-origin	string	This is used to control access from client web (XHR) to Diffusion. This element will enable Cross Origin Resource Sharing. This is a REGEX pattern that will match the origin of the request. A value of ".*" matches anything so all requests are allowed. If not specified then the service will not be able to handle CORS requests.	[0 .. 1]
websocket-secure-response	boolean	Indicate that the websocket response will state that it is from a secure connection, when it is not. You would use this option, if SSL off-loading was enabled on a load-balancer. If not specified then this is false.	[0 .. 1]
close-callback-requests	boolean	For Diffusion client requests this specifies whether to obey the keep alive header or close all requests. If true then all all requests will be closed. If not specified then false is assumed.	[0 .. 1]
compression-threshold	bytes	Enable compression for HTTP Client responses over this size. If not specified then 512 is assumed.	[0 .. 1]
max-inbound-request-size	bytes	The maximum number of bytes that the HTTP request can have. If this is not specified then the maximum message size is assumed.	[0 .. 1]
comet-bytes-before-new-poll	bytes	This parameter allows you to specify the number of bytes after which a Comet connection is forced to re-establish itself. This can help to reduce the potential for memory leaks in the browser due to the long-	[0 .. 1]

Copyright 2013 Push Technology

Name	Type	Description	Range
		lived nature of a Comet connection, at the expense of degraded performance due to more frequent HTTP handshakes. If this is not specified then 30k is assumed.	
comet-initial-message-padding	bytes	Some browsers do not pass on data received via a Comet connection until a minimum number of bytes have been received. This means that in the case where an initial load message is small, the client may never receive it. In order to work around this restriction, you can set a value here which will ensure that the first message received will be padded with extra bytes that are automatically discarded by the client library. If not specified then a value of 1k is assumed.	[0 .. 1]

http-service

HTTP Service.

Attributes

Name	Type	Description	Required
name	string	HTTP Service Name.	true
debug	boolean	Set true to debug the HTTP service. If not specified then false.	false

Name	Type	Description	Range
class	string	The user HTTP service class name. This class must implement the HTTPServiceHandler interface in the Web Server API.	[1 .. 1]
url-pattern	string	The pattern that the URL must match for this service to be invoked.	[1 .. 1]
log	string	An optional log file can be specified, and if so, HTTP	[0 .. 1]

Copyright 2013 Push Technology

Name	Type	Description	Range
		access can be logged. The log definition must exist in Logs.xml.	
max-inbound-request-size	bytes	The maximum number of bytes that the HTTP request can have. If this is not specified then the maximum message size is assumed.	[0 .. 1]
property	*property*	HTTP Service Property.	[0 .. -1]

property

HTTP Service Property.

Attributes

Name	Type	Description	Required
name	string	Property Name.	true
type	string	Optional Property Type.	false

file-service

Optional File Service.

Attributes

Name	Type	Description	Required
name	string	File Service Name.	true

Name	Type	Description	Range
virtual-host	*virtual-host*	Virtual Host.	[1 .. -1]
write-timeout	millis	Write timeout for serving files, this does not affect HTTP Clients. If not specified then 3s is assumed.	[0 .. 1]

virtual-host

Virtual Host.

Attributes

Name	Type	Description	Required
name	string	Virtual Host Name.	true
debug	boolean	Debug flag. Set to true for debugging. Default is false.	false

Copyright 2013 Push Technology

Name	Type	Description	Range
host	string	Specifies the host of which the virtual host is to serve, i.e. download.pushtechnology.com or * for all.	[1 .. 1]
document-root	string	The physical directory for this virtual host.	[1 .. 1]
home-page	string	The default home page, this file is used with directory browsing.	[1 .. 1]
error-page	string	This is used to control the 404 response. The server will look for one of this files in the directory of the request, if the file does not exist then it will look for this file in the virtual directory. If no supplied or the file does not exist a standard 404 response HTML document will be sent.	[0 .. 1]
static	boolean	If this is set to true, then after loading the resource once, the file system will not be checked again. This improves performance for simple static usage. By default this is false.	[0 .. 1]
minify	boolean	Set to true to minify html. This will happen before the file is compressed. By default this is false.	[0 .. 1]
cache	*cache*	The Virtual Host Cache Configuration.	[1 .. 1]
compression-threshold	bytes	All HTTP responses over this size will be compressed. If not specified then 512 is assumed.	[0 .. 1]
alias-file	string	Optionally specifies an alias file, this allows for URL aliasing if required.	[0 .. 1]
realms	*realms*	Virtual Host Realms.	[0 .. 1]

cache

The Virtual Host Cache Configuration.

Copyright 2013 Push Technology

Attributes

Name	Type	Description	Required
debug	boolean	Set true to debug the cache. If not specified then false.	false

Name	Type	Description	Range
file-size-limit	bytes	If the file to be served is over this size, then do not cache the entire contents, but map the file instead. If not specified then 1m is assumed.	[0 .. 1]
cache-size-limit	bytes	Total size of the cache for this Web Server definition. If not specified then 10m is assumed.	[0 .. 1]
file-life-time	millis	If the file has not been accessed within the time specified then remove the entry from the cache. If not specified then 1d is assumed.	[0 .. 1]

realms

Virtual Host Realms.

realm

A Virtual Host Realm.

Attributes

Name	Type	Description	Required
name	string	Virtual Host Realm Name.	true
path	string	Virtual Host Realm Path.	true

users

Virtual Host Realm Users.

user

Virtual Host Realm User.

Attributes

Name	Type	Description	Required
name	string	Virtual Host Realm User Name.	true

Copyright 2013 Push Technology

Name	Type	Description	Required
password	string	Virtual Host Realm User Password.	true

logs

Logs.xml - Properties defining Logging options.

logs

Properties defining Logging options.

Name	Type	Description	Range
console-log-level	log-level	The log level to start console logging at. Can be SEVERE, WARNING, ADVICE, INFO, FINE or FINEST. If not specified then INFO is assumed.	[0 .. 1]
log-message-data	boolean	Indicates whether the data part of Messages should be logged as part of routine diagnostic message logging (FINE and FINEST levels). If this is false then credentials message headers will also be hidden. If not specified then true is assumed.	[0 .. 1]
server-log	string	The log to use for the server. This must specify the name of a configured log definition.	[1 .. 1]
default-log-directory	string	The default log folder for all logs, although this can be over-ridden for each log.	[1 .. 1]
async-logging	boolean	Indicates whether logging is asynchronous, that is performed by a separate thread as opposed to being performed in line by the logging thread. This should normally be set to true for performance reasons, but it has been known to cause problems in some OS environments and so this provides the option to turn it off if so advised. If not specified then true is assumed.	[0 .. 1]

Copyright 2013 Push Technology

Name	Type	Description	Range
logging-queue-size	positiveInt	The size of the asynchronous logging queue. This should normally be left at the default value of 128k entries.	[0 .. 1]
thread-name-logging	boolean	Indicates whether the thread name should be logged with messages. If not specified then thread names will be logged.	[0 .. 1]
log	*log*	A Log Definition.	[0 .. -1]

log

A Log Definition.

Attributes

Name	Type	Description	Required
name	anySimpleType	Name of the Log Definition	true

Name	Type	Description	Range
log-directory	string	The name of the directory to which this log file will be written. If not specified then the default-log-directory is assumed.	[0 .. 1]
file-pattern	string	This is used to specify the name of the Log file. The following values may be used within the pattern. "/" - the local pathname separator. "%t" - the system temporary directory. "%g" - the generation number to distinguish rotated logs. "%h" - the value of the "user.home" system property. "%u" - a unique number to resolve conflicts. "%s" - System type - e.g. 'Diffusion'. "%n" - System name as defined in Server.xml. "%d" - Date as specified in diffusion.properties (date.format), this should be included when using daily rotation. "%%"	[0 .. 1]

Copyright 2013 Push Technology

Name	Type	Description	Range
		- translates to a single percent sign "%". If this is not specified then "%s.log" is assumed.	
level	log-level	Specifies the starting log level. Can be SEVERE, WARNING, ADVICE, INFO, FINE or FINEST. If not specified then ADVICE is assumed.	[0 .. 1]
xml-format	boolean	Indicates whether the log file should be output in XML format. If not specified then fasle is assumed.	[0 .. 1]
file-limit	bytes	Specifies an approximate maximum amount to write (in bytes) to any one log file. If this is zero, then there is no limit. If not specified then 0 is assumed.	[0 .. 1]
file-append	boolean	Specifies whether log records should be appended to existing log files. If false then files are overwritten. If not specified then false is assumed.	[0 .. 1]
file-count	positiveNonZeroInt	Specifies the number of log files to use. Must be at least 1. If not specified then 1 is assumed.	[0 .. 1]
rotate-daily	boolean	Indicates whether the log is to rotate on a daily basis. If not specified then the log is not rotated.	[0 .. 1]

management

Management.xml - Specifies System Management Properties.

management

The Management Information

Attributes

Name	Type	Description	Required
enabled	boolean	Specifies if JMXRMI services are started, making JMX remotely available.	true

Copyright 2013 Push Technology

Name	Type	Description	Range
host	string	The management host (RMI registry host). If not specified then "localhost" is assumed.	[0 .. 1]
registry-port	port	The RMI Registry port. If not specified then 1099 is assumed.	[0 .. 1]
connection-port	port	The JMXRMI connection port. If not specified then 1100 is assumed. This relates to the normally ephemoral port employed by JMXRMI. This is normally only useful when configuring firewalls.	[0 .. 1]
register-topics	boolean	Deprecated in favour of features in Statistics.xml. Enable registration of Topics to the JMX server. If you want Topic level statistics via JMX then this should be set to true. If not specified then false is assumed.	[0 .. 1]
users	*users*	The Management Users.	[1 .. 1]

users

The Management Users.

user

A User that can use the JMX interface.

Name	Type	Description	Range
name	string	User name for JMX credentials.	[1 .. 1]
password	string	Password for JMX credentials.	[1 .. 1]
read-only	boolean	Specify if the user has read only access, if this value is false, then the user will have admin access.	[1 .. 1]

statistics

Statistics.xml - properties defining statistics collection. The statistics are broken into sections: client, topic, server and publisher.

Copyright 2013 Push Technology

statistics

Properties defining statistics collection.

Attributes

Name	Type	Description	Required
enabled	boolean	A global switch to toggle collection of all statistics	true

client-statistics

Optional Client Statistics: configures Diffusion to periodically output client statistics to a log file defined in Logs.xml It will give a count of all of the different client types. Each counter is reset according to the configured frequency.

Attributes

Name	Type	Description	Required
enabled	boolean	Specifies if aggregate Client statistics are enabled.	true

Name	Type	Description	Range
log-name	string	Definition of the log in Logs.xml.	[0 .. 1]
output-frequency	millis	Specifies the output frequency of the log, there will be one entry per specified interval. If this is not specified then 1h is assumed.	[0 .. 1]
reset-frequency	millis	Specifies when the counters are reset. Zero means that the counters are never reset. If this is not specified then 1h is assumed.	[0 .. 1]
monitor-instances	boolean	Specifies if individual Client statistics are enabled .	[0 .. 1]

topic-statistics

Optional Topic Statistics.

Attributes

Name	Type	Description	Required
enabled	boolean	Specifies if aggregate Topic statistics are enabled.	true

Copyright 2013 Push Technology

Name	Type	Description	Range
monitor-instances	boolean	Specifies if individual Topic statistics are enabled.	[0 .. 1]

server-statistics

Optional Server Statistics.

Attributes

Name	Type	Description	Required
enabled	boolean	Specifies if Server statistics are enabled. This enables high-level aggregate statistics for the system.	true

Name	Type	Description	Range
monitor-instances	boolean	Specifies if individual Event Publisher statistics are enabled.	[0 .. 1]

publisher-statistics

Optional Publisher Statistics.

Attributes

Name	Type	Description	Required
enabled	boolean	Specifies if aggregate Publisher statistics are enabled.	true

Name	Type	Description	Range
monitor-instances	boolean	Specifies if individual Publisher statistics are enabled.	[0 .. 1]

reporters

Optional set of StatisticsReporters to be loaded with Diffusion, which will be registered with the internal StatisticsService and used to generate output.

reporter

A Reporter definition.

Attributes

Name	Type	Description	Required
enabled	boolean	Whether the reporter is enabled. If set to true, the reporter will be automatically loaded when	true

Copyright 2013 Push Technology

Name	Type	Description	Required
		Diffusion starts. Otherwise, you will have to manually load the reporter config at run-time using the Statistics API.	
name	string	The Reporter name.	true

Name	Type	Description	Range
type	string	The type of Reporter to be used. Currently options are: TOPIC; exposes metrics in the Diffusion topic tree. JMX; exposes metrics on the local jmx server.	[1 .. 1]
property	*property*	A Property available to the Reporter. This can be used to configure Reporter specific variables or parameters.	[0 .. -1]

property

A Property available to the Reporter. This can be used to configure Reporter specific variables or parameters.

Attributes

Name	Type	Description	Required
name	string	The Property Value	true
type	string	An optional property type. Usage of this is implementation specific.	false

connection-validation-policies

A Connection Validation Policy file.

connection-validation-policies

Connection Validation Policies

policy

A Connection Validation Policy.

Attributes

Name	Type	Description	Required
type	string	The policy type should be either "blacklist" or "whitelist". A blacklist indicates that if any of the policy rules in	true

Copyright 2013 Push Technology

Name	Type	Description	Required
		this policy match the incoming connection, then that connection is to be rejected. A whitelist requires that at least one policy rule matches for the connection to be accepted.	
name	string	Each policy must be supplied with a unique name for easy reference.	true
automatic	boolean	Policies which are set to automatic are applied by Diffusion and the publishers do not need to perform any checks themselves. If this attribute is set to false, then the policy is not applied unless it is done so manually by the publisher. If not specified then true is assumed.	false

addresses

Connection Validation Policy Addresses. These are addresses that are blacklisted/whitelisted.

Name	Type	Description	Range
address	string	An IP address (or regex) to match against a connecting client.	[0 .. -1]
hostname	string	The hostname (or regex) of a connecting client.	[0 .. -1]
resolved-name	string	The resolved hostname (or regex) of a connecting client, as returned by the WhoIs service.	[0 .. -1]

locale

Connection Validation Policy Locale. This is locale details that are blacklisted/whitelisted.

Name	Type	Description	Range
country	string	The ISO country code of the connecting client, as returned by the WhoIs service.	[0 .. 1]
language	string	The ISO language code of the connecting client,	[0 .. 1]

Copyright 2013 Push Technology

Name	Type	Description	Range
		as returned by the WhoIs service.	

subscription-validation-policies

A Subscription Validation Policy file.

subscription-validation-policies

Subscription Validation Policies

topics

A map of Topics to Policies.

topic

A topic to Policy mapping.

Attributes

Name	Type	Description	Required
policy	string	The name of the policy to apply to this topic.	true

policy

A Subscription Validation Policy.

Attributes

Name	Type	Description	Required
type	string	The policy type should be either "blacklist" or "whitelist". A blacklist indicates that if any of the policy rules in this policy match the incoming connection, then that connection is to be rejected. A whitelist requires that at least one policy rule matches for the connection to be accepted.	true
name	string	Each policy must be supplied with a unique name for easy reference.	true
validate-children	boolean	Controls whether to perform validation on child topics if the parent topic fails validation. If not specified then false is assumed.	false

Copyright 2013 Push Technology

Name	Type	Description	Required
automatic	boolean	Policies which are set to automatic are applied by Diffusion and the publishers do not need to perform any checks themselves. If this is set to false, then the policy is not applied unless it is done so manually by the publisher. If not specified then true is assumed.	false

addresses

Subscription Validation Policy Addresses. These are addresses that are blacklisted/whitelisted.

Name	Type	Description	Range
address	string	An IP address (or regex) to match against a subscribing client.	[0 .. -1]
hostname	string	The hostname (or regex) of a subscribing client.	[0 .. -1]
resolved-name	string	The resolved hostname (or regex) of a subscribing client, as returned by the WhoIs service.	[0 .. -1]

locale

Connection Validation Policy Locale. This is locale details that are blacklisted/whitelisted.

Name	Type	Description	Range
country	string	The ISO country code of the subscribing client, as returned by the WhoIs service.	[0 .. 1]
language	string	The ISO language code of the subscribing client, as returned by the WhoIs service.	[0 .. 1]

env

Env.xml - Environment Variables used in configuration.

env

property

Environment Variable Value

Attributes

Copyright 2013 Push Technology

Name	Type	Description	Required
name	token	Name of the environment variable.	true

aliases

Applies to an Aliases file used in a Web Server.

aliases

List of Aliases

alias

An Alias Definition

Attributes

Name	Type	Description	Required
name	string	A name for the alias	true

Name	Type	Description	Range
source	string	The source URL, which can be expressed as a regular expression	[1 .. 1]
destination	string	The destination path	[1 .. 1]

mimes

Mimes.xml - Mime types

mimes

mime

Mime.

Attributes

Name	Type	Description	Required
type	string	Mime Type.	true
extension	string	Mime Extension.	true

Programmatic Configuration

Configuring Diffusion™ programmatically

An alternative to configuring a Diffusion™ Server using XML property files is to instantiate a Diffusion™ Server within a Java application and configure it programmically before starting it.

Using this technique it is possible to do without XML property files altogether as every aspect of the XML configuration can also be supplied programmatically.

Copyright 2013 Push Technology

If desired, some properties may be loaded from XML files and some supplied programmatically or default properties can be bootstrapped from XML files and overridden programmatically before the Server is started.

Most Server properties may only be configured before the Server is started, thus the need to instantiate the Server within an application and configure before starting the Server. However, certain configuration items (examples being Conflation and Connection Policies) can be configured at any time during the life of the server. The Javadoc will make it clear if a property can be changed at runtime.

Because the properties that can be set programmatically refelect those that can be set in XML this section will not describe the properties in detail. The XSD property descriptions or the Javadoc for the configuration API can be consulted for full details.

As well as allowing configuration properties to be set the Confguration API also allows all properties that can be configured to be read at runtime. So Publisher code has direct access to all property settings.

Using the Configuration API

General Use

From within a Java application the root of the *configuration tree* can be obtained at any time using `ConfigManager.getConfig()`. This will provide access to the general objects and can be used from within server side or client side code.

From within server side code (e.g a Publisher) the Server configuration root can be obtained using `ConfigManager.getServerConfig()` which exposes all of the server side configuration also.

From the configuration root you can navigate to any subordinate configuration objects to view them or set their properties.

On the server side most properties cannot be changed after the Server has started and they become 'locked' so any attempt to change them would result in an exception. Certain properties (such as Conflation and Connection Policies) can be changed at runtime. The javadoc will make it clear which properties can be changed at runtime.

In client side Java code the configuration does not become locked and can be changed at any time but it should be noted that some values are only read at the start so ideally all setting of properties should be done before creating any client side objects.

For configuration objects which are optional but there can be many (multiplicity 0..n) then there will be appropriate add methods to add new objects. For example to add a Publisher and set a property on it:-

```
PublisherConfig publisher =
    ConfigManager.getServerConfig().addPublisher(
        "MyPublisher",
        "com.pub.MyPublisher");
publisher.setTopicAliasing(false);
```

In these cases there would also be methods to obtain the full list (e.g. `getPublishers()`) or to obtain a specific one by name (e.g. `getPublisher("MyPublisher")`). In many cases there will also be methods to remove an object.

It should be noted that when there must be at least one object (multiplicity 1..n) then you should allways configure at least one. However, if a server is started with missing configuration of this ind then suitable defaults would normally be created and a warning logged.

Single instance configuration objects (multiplicity 1..1) subordinate to the root can be obtained so that their properties may be changed (or read). So, for example the Queues object (an instance of `QueuesConfig`) can be obtained using the `getQueues()` method.

When a single configuration object is optional (multiplicity 0..1) then the get method could return null if it has not been defined. In this case to set it there would be a set method (as opposed to add) which would return the object

Copyright 2013 Push Technology

created. An example of this is the file service (`FileServiceConfig`) on a Web Server (`WebServerConfig`) as shown in the following example code:-

```
ServerConfig config = ConfigManager.getServerConfig();
WebServerConfig webServer = config.addWebServer("MyWebServer");
FileServiceConfig fileService = webServer.setFileService("File Service");
```

Configuring a Server

After instantiating a Diffusion™ Server in Java the root of the Server Configuration tree can be obtained from the Server object itself and then configuration objects can be navigated to and changed as required before starting the server.

For example the following code shows how to add a connector that accepts client connections on port 9090:-

```
DiffusionServer server = new DiffusionServer();
ServerConfig config = server.getConfig();
ConnectorConfig connector = config.addConnector("Client Connector");
connector.setPort(9090);
connector.setType(Type.CLIENT);
server.start();
```

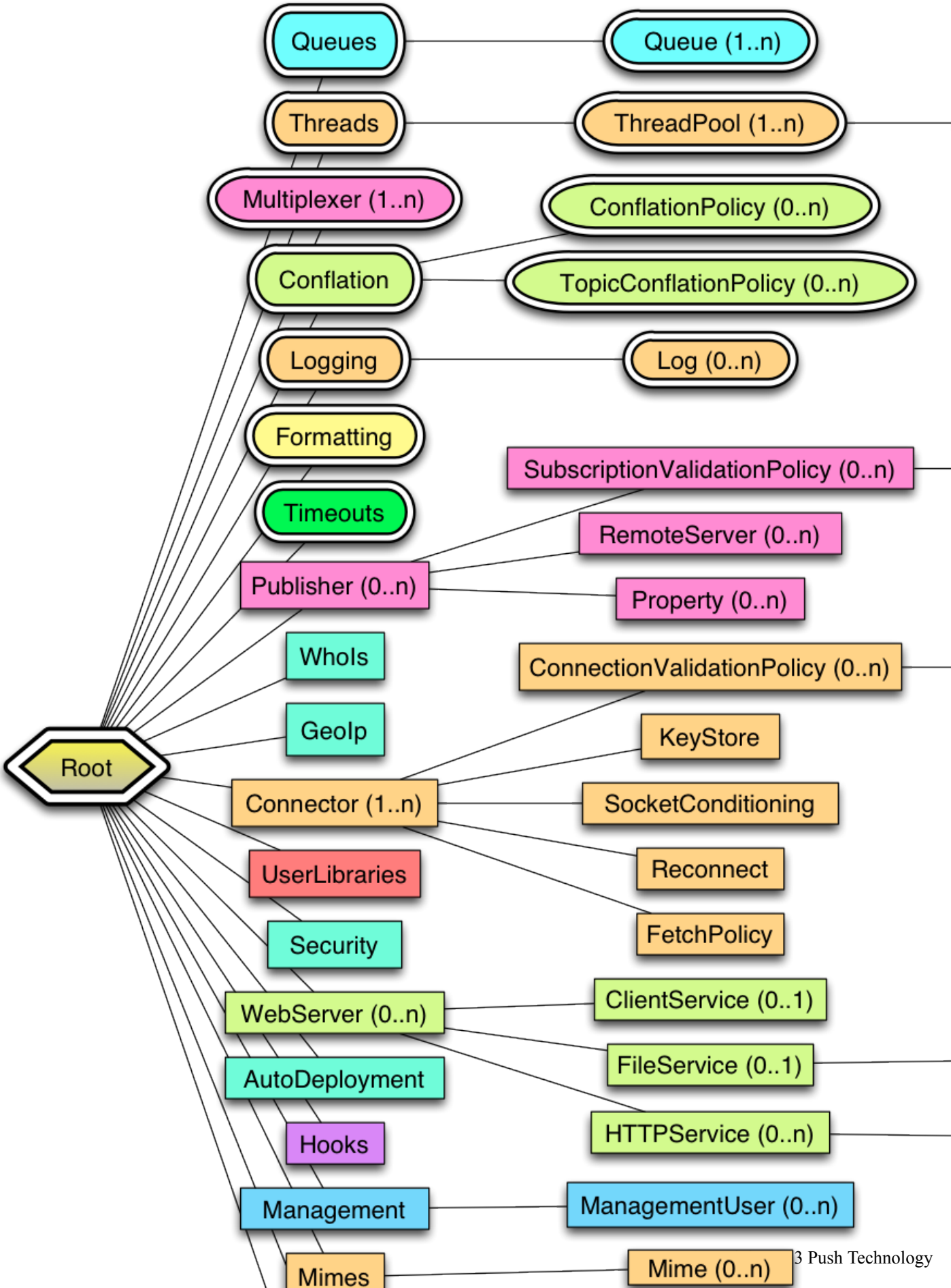
In reality you would probably want to configure far more. However, if any essential objects are ommitted (such as queues) then suitable defaults are created when the server starts and a warning is logged.

Configuration Access from a Publisher

Within a Publisher the configuration object for the Publisher itself can be obtained using the `getConfig` nmethod which will return the Publisher configuration (`PublisherConfig`) object.

Copyright 2013 Push Technology

The Configuration Tree



All general objects can be obtained by navigating from the root object obtained from `ConfigManager.getConfig()`.

Server side objects can only be reached in a server environment using `ConfigManager.getServerConfig()`.

Copyright 2013 Push Technology

Chapter 19

Adapters

Topics:

- *JMS Adapter*

Adapters are connectors to external systems, providing a simplified bridge between data held in those systems and Diffusion.

JMS Adapter

JMS Adapter for Diffusion

Overview

The JMS Adapter for Diffusion™ allows Diffusion clients to transparently send and receive messages with topics and queues on a JMS server. The adapter exposes JMS destinations in a Diffusion topic tree transparently, with no Java coding required, and allows Diffusion clients to both receive and send messages with a JMS server.

Mapping Diffusion topics to JMS topics and queues

Diffusion maintains a topic tree which maps to JMS topics and queues. Each JMS adapter should be configured with a root topic, e.g. `jms/activemq` in order to partition itself from other JMS adapter instances. In `JMSAdapter.xml`, this is set using the `<root-topic>` element.

Underneath the root topic, there are four reserved sub-topics. Given a root topic `jms`, there will be subtopics `topic`, `queue`, `tmp` and `reply`. The `tmp` subtopic also contains `topic` and `queue`, which map to JMS `TemporaryTopic` and `TemporaryQueue` destinations respectively.

The `reply` topic is reserved for subtopics associated with request-reply messages originating from another client of the JMS server. An example of how this is used can be found in the examples below.

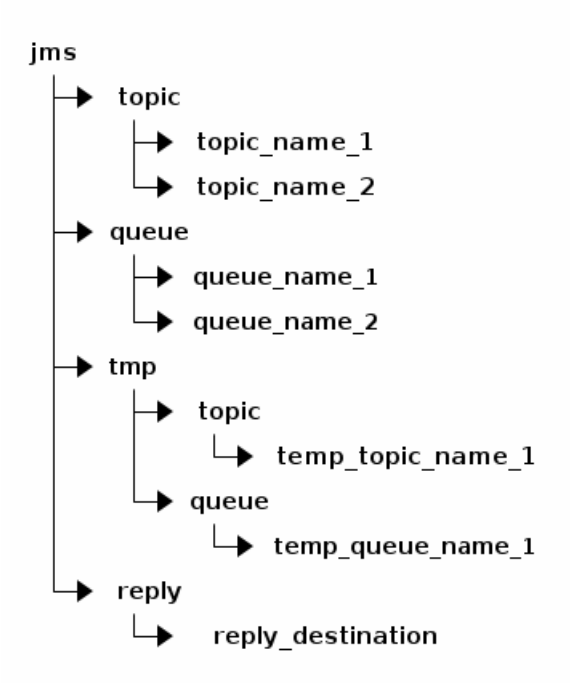


Figure 55: JMS adapter topic tree layout

When a client subscribes to the Diffusion topic, `jms/activemq/topic/ABC` that has not previously been subscribed to, Diffusion will attempt to transparently connect to the matching JMS topic and any messages that Diffusion receives from the JMS server are relayed to the Diffusion client. At present, it is not possible to subscribe to JMS topics using wildcards.

Note that Diffusion topics are created dynamically and do not exist until a valid JMS subscription has been made and data received.

Mapping to JMS queues is performed in the same way. Delivery of messages to clients subscribing to the associated Diffusion topic is different, however, and in keeping with the delivery characteristics of JMS queues. When there are multiple Diffusion clients listening for data originating from the same JMS queue each message will be delivered to

Copyright 2013 Push Technology

at most one client. Depending on the configuration in `JMSAdapter.xml` the receiving client will either be chosen randomly, or based on the client with the fewest number of messages waiting for delivery.

Temporary topics and queues

A Diffusion client may request access to a JMS temporary topic (or queue). This is achieved in the same way as subscribing to a JMS destination except that it uses a different part of the Diffusion topic tree. A common use for temporary queues is to set up a return path for request-reply operations. In the topic tree, temporary topics exist as sub-topics under `jms/tmp/topic`.

Message headers

The JMS adapter will copy the standard JMS headers and user-supplied headers when converting between JMS and Diffusion message types. In Diffusion, headers are logically grouped in pairs, where property n is the name, and n+1 is the value (where n is an even number).

In the case of the `JMSReplyTo` header, the related header `DiffusionReplyTo` is created. Mapping between a `JMSReplyTo` destination and a `DiffusionReplyTo` topic is handled transparently by the adapter. In most circumstances, a Diffusion client can ignore the `JMSReplyTo` header; it is forwarded to the client for completeness. Example usage of `DiffusionReplyTo` follows in the examples section.

Acknowledgement modes

The only acknowledgement mode supported is `AUTO_ACKNOWLEDGE`. This imples that when any message is received from the JMS server, an acknowledgement is sent from the JMS adapter to the JMS server. Client-level acknowledgements (`CLIENT_ACKNOWLEDGE`) originating from a Diffusion client are not supported.

Installing the JMS Adapter

To use the JMS adapter for Diffusion, you will need the following:

1. `jmsadapter.jar` in the Diffusion `CLASSPATH` (for example, in the `lib` directory).
2. The JMS library for your specific JMS vendor in the Diffusion `CLASSPATH`.
3. An entry in *`etc/Publishers.xml`* enabling the JMS adapter with a link to the `JMSAdapter.xml` file.
4. A correctly configured `JMSAdapter.xml` file. For validation purposes, the associated XML schema can be found in the file `xsd/JMSAdapter.xsd` in the Diffusion installation directory.

Configuring the JMS Adapter

The configuration file for the JMS adapter is typically called `JMSAdapter.xml`, although this may be overridden by a setting in `Publishers.xml`.

1. Configure `Publishers.xml`

   Instantiate the JMS Adapter via enabling it in `etc/Publishers.xml`, for example:

```
<publisher name="JMSAdapter">
  <class>com.pushtechnology.diffusion.adapters.jms.JMSAdapter</class>
  <enabled>true</enabled>
  <start>true</start>
  <property name="config.filename">../etc/JMSAdapter.xml</property>
</publisher>
```

2. Configure `JMSAdapter.xml`

   A `JMSAdapter.xml` for ActiveMQ may look like this:

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="yes"?>
```

Copyright 2013 Push Technology

```
<jms-config>
  <binding>
    <env>
      <property name="java.naming.factory.initial">
        org.apache.activemq.jndi.ActiveMQInitialContextFactory
      </property>
      <property name="java.naming.provider.url">
 tcp://localhost:61616
      </property>
      <property name="java.naming.security.principal">
 jndi_username
      </property>
      <property name="java.naming.security.credentials">
 jndi_password
      </property>
    </env>
    <connection-factory name="ConnectionFactory">
      <credentials username="username" password="password"/>
      <reconnect>
        <max-reconnections>10</max-reconnections>
        <interval>5000</interval>
      </reconnect>
    </connection-factory>
    <root-topic>jms/activemq</root-topic>
    <priority low="3" high="7" />
    <queue-distribution-mode>SMALLEST_QUEUE</queue-distribution-mode>
  </binding>
  <mapping>
    <artefact-names jms=".$" diffusion="/~"/>
  </mapping>
</jms-config>
```

Similarly, a sample `JMSAdapter.xml` Tibco EMS looks like this:

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="yes"?>
<jms-config>
  <binding>
    <env>
      <property name="java.naming.factory.initial">
        com.tibco.tibjms.naming.TibjmsInitialContextFactory
      </property>
      <property name="java.naming.provider.url">
        tcp://localhost:7222
      </property>
      <property name="java.naming.security.principal">
 jndi_username
      </property>
      <property name="java.naming.security.credentials">
 jndi_password
      </property>
    </env>
    <connection-factory name="ConnectionFactory">
      <credentials username="jms_username" password="jms_password"/>
      <reconnect>
 <max-reconnections>10</max-reconnections>
 <interval>5000</interval>
      </reconnect>
    </connection-factory>
    <root-topic>jms/tibco</root-topic>
    <priority low="3" high="7" />
    <queue-distribution-mode>SMALLEST_QUEUE</queue-distribution-mode>
```

Copyright 2013 Push Technology

```
  </binding>
  <mapping>
    <artefact-names jms=".$" diffusion="/~"/>
  </mapping>
</jms-config>
```

Tibco EMS requires that a `ConnectionFactory` definition is provided in `factories.conf`, for example:

```
[ConnectionFactory]
  type = generic
  url = tcp://localhost:7222
  ssl_verify_host = disable
```

This has been tested with Tibco EMS 7.0; see the Tibco EMS documentation for further details.

For Websphere MQ v7.x, the binding section may look like this:

```
<env>
  <property
 name="java.naming.factory.initial">com.sun.jndi.fscontext.RefFSContextFactory</
property>
  <property name="java.naming.provider.url">file:///var/mqm/jndi</
property>
</env>
```

Websphere MQ can include MQRFH2 headers in messages sent between JMS and non-JMS systems. For control over this behaviour, the property `mq.target.client` may be set in `Publishers.xml`; to disable the headers, set this value to `1`. For the default behaviour, do not provide the property.

```
<property name="mq.target.client">1</property>
```

Some JMS vendors (e.g. Websphere MQ) require a JMS `Session` for each topic or queue subscription. The default configuration for the JMS Adapter does not allow for this, but it may be enabled by setting the `use.global.session` property to `false` in `Publishers.xml`:

```
<property name="use.global.session">false</property>
```

<env>	All properties within the `<env>` tag of `JMSAdapter.xml` are used when creating the `InitialContext` which is in turn used to create the connection to the JMS server.
<connection-factory>	This tag specifies the name of the connection factory to use; this will vary between JMS vendors and the server configuration.
<credentials>	Optionally, specify a username and/or password which is used to create a JMS client connection. However, most JMS implementations are likely to have restricted access for anonymous clients or clients which do not require authentication so you can specify a user here who has the necessary privileges to receive and send messages to the JMS destinations that are to be exposed through Diffusion.
<reconnect>	If a connection cannot be made between Diffusion and the JMS server or the connection is severed, the

Copyright 2013 Push Technology

	<max-reconnections> and <interval> parameters allow you to specify how many times the connection should be retried before giving up and how long to wait between each attempt. A value for <max-reconnections> of -1 indicates that the adapter should keep trying to connect forever.
<root-topic>	In order to provide partitioning of the topic tree between topics related to JMS and other topics, it is necessary for a root topic name is defined here. In the event of more than one JMS adapter, it is also possible to segment the topic tree further.
<priority>	JMS supports messages with up to 10 priority levels (0-9) with 0-4 considered to be different grades of "normal" priority and 5-9 "high" priority. Diffusion only has he concept of low, medium and high priority. Using this parameter, it is possible to map JMS messages within a given priority range to a representative Diffusion priority, and vice-versa.
<queue-distribution-mode>	Unlike topics, a message on a JMS queue must be delivered to only one client. When Diffusion receives a message from a queue, it uses this parameter to determine which of its connected clients that is subscribed to the corresponding Diffusion topic should be selected to receive that message. Valid values are: **SMALLEST_QUEUE** Choose a client with the fewest number of messages outstanding in its message queue from Diffusion. **RANDOM** Select a client randomly.
<artefact-names>	Not all characters in a Diffusion topic names are valid JMS topic or queue names (and vice-versa). The two attributes on this element (jms and diffusion) are lists of characters where the n^{th} character in one will be replaced by the corresponding n^{th} character in the other.

JMS Adapter examples

The following scenarios assume that the JMS adapter is configured and connected to a JMS server with a root-topic of jms.

Receiving messages from JMS

A user wants to receive messages from JMS Topic "XYZ".

1. Diffusion client creates a subscription to the topic jms/topic/XYZ.
2. Once a message has been sent from the source system into the JMS server, the Diffusion client will receive an Initial Topic Load Message.
3. Subsequent messages from the source system result in Delta messages being delivered to the Diffusion client.

The same technique is used for receiving messages from JMS queues, with the following differences:

Copyright 2013 Push Technology

- Other clients subscribing to the same JMS queue (either via Diffusion or directly using JMS) could receive the message instead of our client.
- All messages originating from JMS queues are Initial Topic Load messages. Since a sequence of messages from a JMS queue are unlikely to always be delivered to the same client, the concept of delta messages does not readily apply and the full message state must be supplied every time.

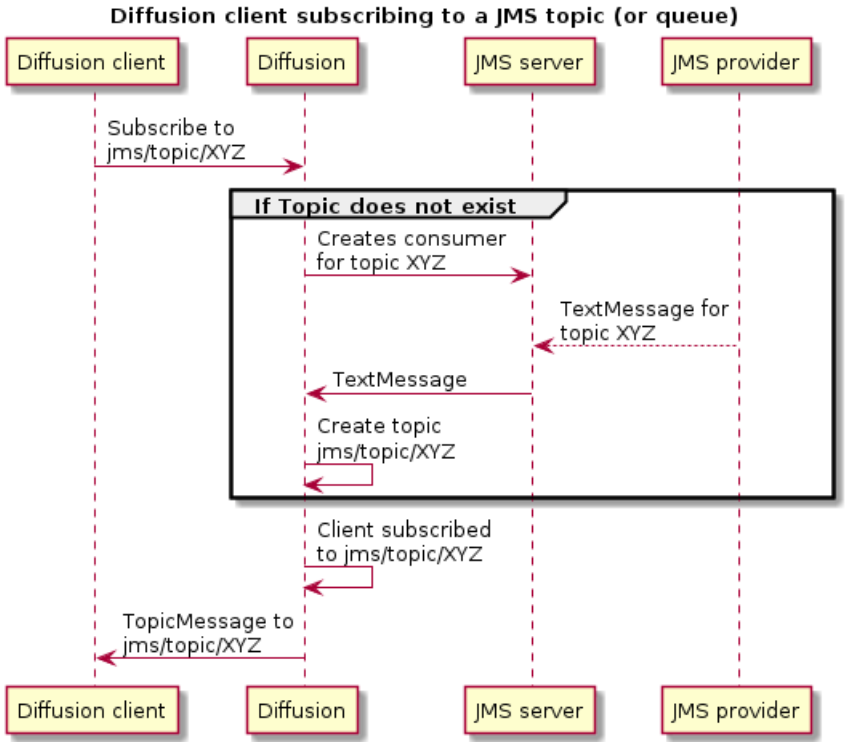


Figure 56: Subscription flow

Sending messages to JMS

A user wants to send messages to the JMS topic "`XYZ`".

1. The Diffusion client sends a message to the topic `jms/topic/XYZ`.
2. The JMS server receives an equivalent TextMessage.

   Unlike most Diffusion solutions, it is not necessary to subscribe to a Diffusion topic before sending it a message which targets a JMS destination.

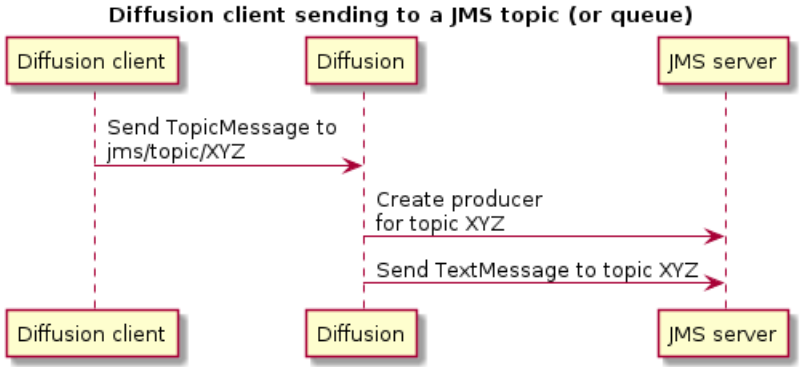


Figure 57: Sending flow

Copyright 2013 Push Technology

Processing a request-reply message with a Diffusion client

A common pattern among JMS solutions is for a client to receive a message from JMS and a reply is expected to be sent to a specific JMS topic or queue. Typically, the JMS publisher creates and sends a message with the `JMSReplyTo` header set to some other destination defined within the JMS server. This could be a topic, queue, or a temporary topic or temporary queue. The Diffusion client does not need to know the destination type as this is handled within the adapter.

1. Diffusion client subscribes to topic `jms/queue/ABC`.
2. JMS producers creates a temporary queue ("`XYZ`") and subscribes to it.
3. JMS producer sends a message to the queue `ABC` with `JMSReplyTo` set to the queue `XYZ`.
4. Diffusion client receives a message on queue `jms/topic/ABC`, with `DiffusionReplyTo` set to `jms/reply/XYZ`.
5. Diffusion client sends a response message to the queue `jms/reply/XYZ`.
6. JMS producer receives a `TextMessage` on the temporary queue `XYZ`.

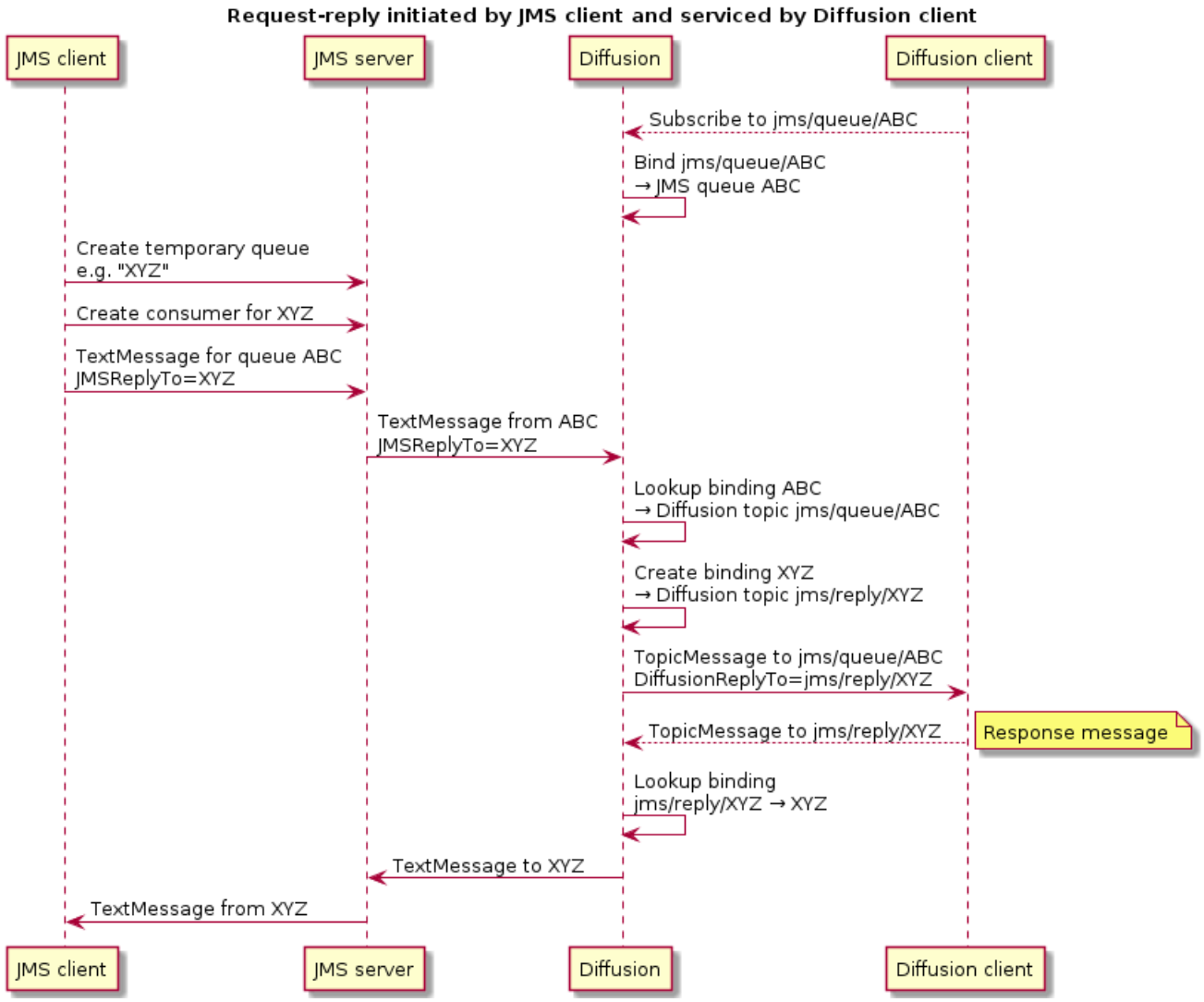


Figure 58: Request-reply from JMS to Diffusion

Sending a request-reply message from a Diffusion-client

It is also possible to send a message from a Diffusion client into a JMS server with the expectation that a JMS client will process the message and send a response back to the same Diffusion client.

Copyright 2013 Push Technology

1. JMS client subscribes to messages on queue `ABC`.
2. Diffusion client subscribes to `jms/tmp/queue/XYZ`. (Commonly, `XYZ` would be a unique identifier).
3. Diffusion client sends a request message to `jms/queue/ABC` with the `DiffusionReplyTo` header set with the value `jms/tmp/queue/XYZ`.
4. JMS client receives the request message on queue `ABC`, with the `JMSReplyTo` header set to queue `DEF`.
5. JMS client sends a reply to queue `DEF`.
6. Diffusion client receives the reply on topic `jms/tmp/queue/XYZ`.

   Note that the return Diffusion topic can be any topic, so it is not necessary that the originating Diffusion client receives the reply - it could be any client listening for messages on that topic.

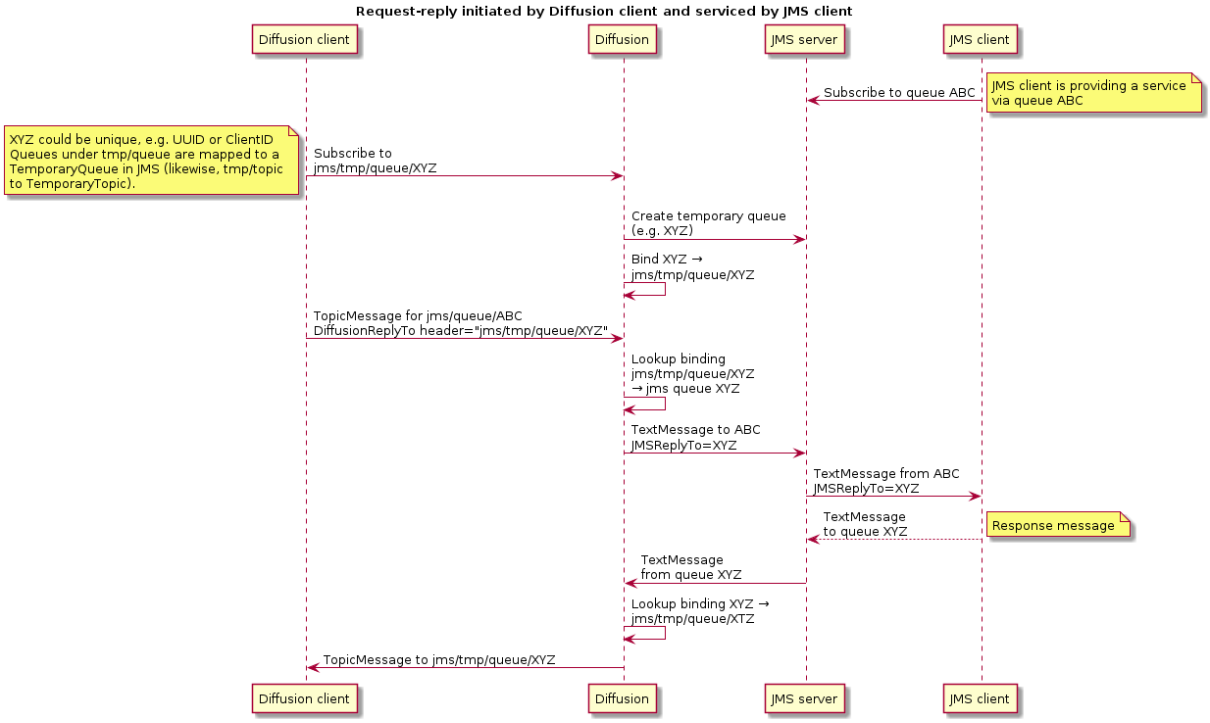

Request-reply initiated by Diffusion client and serviced by JMS client
Diffusion client
Diffusion
JMS server
JMS client
Subscribe to queue ABC
JMS client is providing a service via queue ABC
XYZ could be unique, e.g. UUID or ClientID Queues under tmp/queue are mapped to a TemporaryQueue in JMS (likewise, tmp/topic to TemporaryTopic).
Subscribe to jms/tmp/queue/XYZ
Create temporary queue (e.g. XYZ)
Bind XYZ → jms/tmp/queue/XYZ
TopicMessage for jms/queue/ABC DiffusionReplyTo header="jms/tmp/queue/XYZ"
Lookup binding jms/tmp/queue/XYZ → jms queue XYZ
TextMessage to ABC JMSReplyTo=XYZ
TextMessage from ABC JMSReplyTo=XYZ
TextMessage to queue XYZ
Response message
TextMessage from queue XYZ
Lookup binding XYZ → jms/tmp/queue/XTZ
TopicMessage to jms/tmp/queue/XYZ
Diffusion client
Diffusion
JMS server
JMS client

Figure 59: Request-reply from Diffusion to JMS

Copyright 2013 Push Technology

Copyright 2013 Push Technology

Chapter 20

Tuning

Topics:

- *Buffer Sizing*
- *Message Sizing*
- *Client Queues*
- *Client Multiplexers*
- *Write Selectors*
- *Connectors*
- *Thread Pools*
- *Client Reconnection*
- *Client Failover*
- *Message Conflation*
- *Client Throttling*
- *Memory Considerations*
- *Platform Specific Issues*
- *Publisher Design*

Aspects of tuning Diffusion™ for better Performance or Resilience

This section covers aspects of configuring Diffusion™ in order to achieve higher levels of perfomance and covers some of the more advanced features which enable users to get more out of Diffusion™.

Buffer Sizing

The significance of buffer sizing to Performance

There are a number of places within the configuration of Diffusion™ where buffer sizes need to be specified and getting these right can have a significant impact upon performance.

Connector Output Buffers

An output buffer size needs to be configured for each Connector.

The output buffer size configured for a Connector must be at least as large as the largest message that is expected to be sent to any Client connecting via that Connector. However, the buffer size can be much larger so that the messages can be batched at the Server thus improving performance (see below).

Each connected Client will be assigned a socket buffer of the specified size if possible. A warning will be logged if a smaller socket buffer was allocated than requested.

In addition each Client Multiplexer will have a buffer of the configured size (as a Multiplexer will only be writing to one of it's Clients at any one time). The multiplexer buffer is used to batch messages from the Client queue before writing and thus if the socket buffer does end up being smaller than the configured buffer and the throughput is high then the allocated socket buffer size could become a bottleneck.

Getting the correct output buffer size is vital. Make this too small and the Diffusion™ Server will not be batch and write messages to Clients at optimal rates. Make them too big and extra memory will be consumed.

Note that for maximum performance all Clients should ideally configure their input buffer size to match the Connector's output buffer size.

Client Output Buffers

As at the server, the output buffer sizes in use need to be configured for a Client.

In the Java Client this is specified in the ServerDetails object used to make the connection. As the Java client does not buffer messages then this only needs to be large enough to cater for the largest message size that will be sent to the server.

Publisher Client Output Buffers

A Publisher Client (a connection made from a Publisher to another Diffusion Server) is slightly different from a normal Client in that it *does* queue and buffer messages for sending and therefore it is advantageous to throughput to use a larger output buffer size.

The output buffer size is configured in the server element in Publishers.xml or in the ServerDetails object depending upon how the connection is being made.

Connector Input Buffers

Each Connector also specifies an input Buffer size.

Each Client that connects will be assigned a buffer of this size to receive inbound messages and therefore this buffer needs to be as large as the largest message expected.

This size will also be used to allocate a receive socket buffer for the Client. It is possible that the socket buffer allocated could actually be less than requested in which case a warning would be logged.

For maximum performance the size used for this buffer should match up with the output buffer size used by Clients.

Client Input Buffers

Clients also need to specify the buffer size to use for input.

Copyright 2013 Push Technology

In the Java client this is specified in the ServerDetails object used to make the connection (or possibly Publishers.xml for a Publisher Client connection).

Matching Buffer Sizes

For optimal throughput it is desirable to match the size of buffers at each end of every connection. The input buffer size used by clients should therefore ideally match the output buffer size at the Connector that they connect to. Also the output buffer size specified by Clients should match the input buffer size of the Connector they connect to.

It should be noted that because Publisher Server connections queue and batch mesages at both ends then it is recommended that a separate Connector is used for such connections such that optimal buffer sizing can be achieved.

Message Batching

The size of output buffers configured for a Connector may be much larger than the largest expected Message size because the Server will use these buffers to batch Client Messages which will improve performance. Ideally the buffer size would be a multiple of the average Message size. Note that when using for HTTP Clients you should allow between 20 (for HTTP Duplex) and 250 (for Browser) bytes extra for control information.

Each Client Multiplexer will assign an output buffer of each size specified by Client connectors. So if there were 3 Client Connectors, each specifying a different output buffer size, and 2 Client Multiplexers then each Multiplexer would assign 3 different buffers (6 in total). When using HTTP Duplex connections then each Multiplexer would assign an additional buffer for chunked encoding.

When a Client Multiplexer is unable to write the contents of an output buffer to a Client in one go then the writing is deferred and the Multiplexer will take a copy of the remaining data in the output buffer into its own temporary buffer.

Message Sizing

The effect of Message Sizing upon Performance

The sizing of Messages that are sent to clients is very important to the overall performance and this must be carefully considered within the design of Publishers.

Every Topic Message has a fixed header of 6, 4 or 3 bytes (depending upon the value of 'Message Length Size") . It then has the Topic Name terminated by one byte, plus any user header information that is also included with the Message.

It is important to work out the size of the Message so that the Connector buffers can be set correctly, otherwise Diffusion™ will be unable to put the Messages on the wire quickly enough.

Byte Pinching

With any messaging system, the smaller the messages, the lower the latency and the faster the system will perform. There is a consultancy exercise that Push Technology performs as a service to analyse the messages and reduce them as much as possible. Best practices here to name but a few are :

- Only send data that is required by the client.
- Look at the data format and strip any fat off the message.
- Is the information being sent a true delta?
- Client side data models

Message Encoding

If you are sending large messages, then it is worth compressing the messages. This will only happen once on the server, and then the clients have the technology to decompress them, this also includes DHTML clients. If other encoding is used then it is worth bearing in mind the CPU overhead required.

Copyright 2013 Push Technology

Client Queues

Tuning Client Queue depths

A 'Maximum Queue Depth' can be configured for Client queues so that Clients are closed if their Message backlog becomes too large. This size must be chosen carefully as a large size may lead to excessive memory usage and vulnerability to 'Denial of Service' attacks, whilst a small size can lead to slow Clients being disconnected too frequently.

Client Queues do not take any memory per se, as Diffusion™ uses a Zero Copy paradigm, but there are consequences in setting them too small or too large. If the Client queue is set too small, once the Client has filled its queue the Diffusion™ Server will close the Client.

When considering queue depth take into account the average message size and publication rate. Messages that are held in the Client queue are not garbage collected and may get promoted, which increases their impact on GC pressure. Should messages in the client queue build up, consider the maximum delay in the context of you application. For example: Assuming 100 bytes is the average message size and the application is publishing an average of 100 messages per second. If the client queue is setup to have a maximum depth of 1000 messages this means we allow messages to build up for a slow client for up to 10 seconds, during this time a slow client will be building up a cache of 100k of messages to be sent. Note that it is natural for queues to build up a little with spikes in publication rate or momentary bandwidth limits, but the tolerance to such delays is expressed in the client queue depth and should be considered in that context.

Client Multiplexers

Tuning Multiplexers for Optimal Performance

The load of batching, conflating and merging Messages being sent from publishers to outbound Clients will be spread across Client Multiplexers. The number of configured Client Multiplexers should take into account the expected Message load and concurrent client connections. The more clients are assigned to a multiplexer the more load it will need to contend with.

A Client multiplexer will process all Client messages into the client queue. Clients are added to the multiplexers according to a load balancing policy (deafult is round robin, but least clients is also supported).

Publishers either 'broadcast' on a topic to all subscribed clients or send clients direct messages. When broadcasting all Multiplexers are notified and go on to find all subscribed Clients which are assigned to the particular Multiplexer. When a message is sent to a particular Client only that Client's Multiplexer is notified. It is therefore more efficient to 'broadcast' than it is to send the same message to a large number of clients by iterating over them.

Client Multiplexers are non-blocking, high priority threads so having too many may be detrimental, as they will be competing for the same resource (CPU). As a rule of thumb, the number of Multiplexers should not exceed the number of available logical cores. If a Client Multiplexer should become over subscribed message latency may increase. For maximum throughput the number of multiplexers could be set to the number of available cores, but this configuration is only recommended in the case where other threads are assumed to be mostly idle (e.g. little inbound traffic, low publisher overhead).

Client Multiplexers performance is influenced by the use of merge and conflation policies as those are executed in the multiplexer thread. It is recommended that conflation policy changes and in particular changes to merge conflation logic be profiled and written with performance in mind. In particular the use of locks or any other blocking code is highly discouraged.

Each Multiplexer will use a different buffer for each output buffer size that is specified to any Connector. So if there were three connectors with different output buffer sizes specified then each multiplexer would assign three different buffers. Each Multiplexer may also assign an extra buffer for HTTP use. A larger output buffer will enable more effecient batching of messages per write, as large writes are generally more effeicient but care must be taken to not overwhelm client connections regularly and causing them to be blocked for any period of time.

Copyright 2013 Push Technology

When a Multiplexer is unable to write a message to a Client because the buffer has become full, a Write Selector will be notified. The Write Selector is responsible for watching the client and notifiying the Multiplexer when it becomes writable. The Muliplexer remains responsible for writing the message.

Write Selectors

Configuring Write Selectors

Write selectors are used to finish writing partially written buffers being sent to Clients by the mulitplexer.

Each connected Client creates a message channel with the Diffusion server. Each message channel has a socket buffer of configured size. The socket buffer fills as the server sends messages to the Client and it empties as messages are written onto the wire. Since Diffusion uses a non-blocking IO model when the socket buffer is full processing will continue if not all of the messages can be written to a single channel. This allows the mulitplexer to send messages to other Clients. The write selectors handle any remaining bytes that need to be sent to the Client.

Each write selector is a separate thread that is notified when a partial write happens. When a selector is notified of a partial write the write selector will monitor the message channel and when it becomes writable again it will notify the multiplexer that it can write more bytes. The multiplexer will write as many of the bytes as it is able to. Write selectors are non-blocking threads so having too many may be detrimental.

The write selectors are configured together as a pool, different write selectors cannot use different configuration values.

You should refer to the *XML Configuration* for information on configuring from the XML files.

You should refer to the WriteSelectorConfig JavaDoc for information on configuring progammatically.

Normally the configured values would be sufficient but when a large number of slow consuming Clients (e.g. SSL) are expected then it may be necessary to increase the size of the pool. The Write Selectors are only used to notifiy the Mulitplexers about slow clients, the Multiplexers are more important to performance.

Connectors

Tuning Connectors

Configuring multiple connectors

When there is more than one Publisher application running on a Server it is likely that a separate Connector should be configured for each one so that buffer requirements etc can be specific to the Connector.

It may also be beneficial to configure different connectors for different client types as their requirements may be different. This is especially true for Publisher Clients where there are low numbers of connections which would benefit from very large buffer sizes in both directions.

Buffers

As Diffusion™ can have tens of thousands of connections at any one time on a machine it is important to make sure that the buffers are set correctly.

In order to reduce the memory footprint, the Diffusion™ Server will condition the input and output buffers. If the buffers are set too small then Diffusion™ will not be able to write the Messages in one go and delegate the task to a writer selector.

There is a separate *section* devoted to tuning buffers.

Copyright 2013 Push Technology

Acceptors

The Acceptor has the job of accepting the open connection from the Client. Having too few of these will mean that the socket server will sustain a backlog and the Client will take time to connect or even timeout. Each Acceptor is a thread so having too many will drain resources from the system.

Thread Pools

What Thread Pools are used for and How to configure them

Thread Pools are used within Diffusion™ to optimise the use of threads.

It is important to understand balance when tuning thread usage for a system. There must be sufficient thread resources required but not so many as to starve other parts of the system. At the end of the day there are only so many threads that a system can provide.

In general when provisioning threads we need to separate the blocking and non-blocking activities. While it is beneficial to have more threads than cores for blocking tasks it is detrimental to the server if more threads than cores are runnable at any given time.

There are a number of places where thread pools are used within Diffusion™. These are disccused in the *server concurrency section*.

Configurable Properties

The following key values may be configured for a thread pool to influence its behaviour and use of resource:-

Property	**Usage**
Core Size	The core number of threads to have running in the thread pool. Whenever a thread is required a new one will be created until this number is reached, even if there are idle threads already in the pool. After reaching this number of threads then at least this number of threads will be maintained within the pool.
Maximum Size	The maximum number of threads that may be created in the thread pool before tasks are queued. If this is specified as 0 then the pool is unbounded and so the task queue size value will be ignored. Generally an unbounded pool is not recommended as it could potentially consume all machine resources.
Queue Size	The pool queue size. When the maximum pool size is reached then tasks will be queued. If the value is zero then the queue is unbounded. If not zero then the value must be at least 10 (it will be automatically adjusted if it is not).
Keep Alive Time	The time limit for which threads may remain idle before being terminated. If there are more than the core number of threads currently in the pool, after waiting this amount of time without processing a task, excess threads will be terminated. A value of zero (the default) will cause excess threads to terminate immediately after executing tasks.
Notification Handler	A thread pool may have a 'notification handler' associated with it to handle certain events relating to the pool. This allows for user written actions to be performed (e.g. sending an email) when certain pool events (like too much task queueing) occur. See below for more details.

Copyright 2013 Push Technology

Property	Usage
Rejection Handler	A thread pool may have a 'rejection handler' associated with it to handle a runnable task that has been rejected. This allows user written actions to handle a runnable task that can not be executed by the thread pool. See below for more details.

Notification Handler

A Thread Pool Notification Handler may be configured to act upon certain Thread Pool events.

These events are:-

Upper Threshold Reached	A specified upper threshold for the pool has been reached. This means the pool size has reached the specified size. The event is notified only once and will not be notified again until the Lower ThresholdReached event has occurred.
Lower Threshold Reached	A specified lower threshold for the pool has been reached after an Upper Threshold Reached event has been notified. This means the pool size has now shrunk the specified size.
Task Rejected	The pool has rejected a task because there are no idle threads available and the task queue has filled. What happens to the rejected task depends upon te type of pool- typically the task is run within the thread that passes the task to the pool which is not desirable which is why it ought to be notified when it occurs. This differs from the rejection handler in that it does not expose the runnable task. This means it can be used only for notification.

The notification handler is a user written class which must implement the `ThreadPoolNotificationHandler` interface in the threads Java API. The name of such a class may be configured for in-bound or out-bound thread pools or for connector thread pools in which case an instance of the class will be created (and thus must have a no arguments constructor) when the thread pool is created.

Rejection Handler

A thread pool may have a 'rejection handler' associated with it to handle a runnable task that has been rejected.

Two rejection handlers are provided with Diffusion™. These are the `ThreadService.CallerRunsRejectionPolicy` and `ThreadService.AbortRejectionPolicy`.

The `ThreadService.CallerRunsRejectionPolicy` will execute the runnable task in the thread that tried to pass the runnable task to the thread service. The `ThreadService.CallerRunsRejectionPolicy` will not execute the task and will instead generate an exception.

Note that the Inbound Thread Pool has a mandatory 'Caller runs' rejection policy.

The rejection handler is a user written class which must implement the `ThreadPoolRejectionHandler` interface in the threads Java API. The name of such a class may be configured for in-bound or out-bound thread pools or for connector thread pools in which case an instance of the class will be created (and thus must have a no arguments constructor) when the thread pool is created.

Adjusting the configuration

Thread pools should be adjusted gradually. Ideally you should be able to duplicate expected maximum loads in test environment. This environment can be used to tune the thread pools to satisfy the load. They should just be able to cope with the maximum load, increasing them beyond this may degrade overall performance.

Background thread pool:

Copyright 2013 Push Technology

In general the defaults should suffice for the tasks assigned to it by Diffusion. If you assign tasks to the pool yourself you may wish to consider, increasing the number of threads.

Inbound thread pool:

This pool is used to handle inbound connections and messages. Increasing the thread pool allows new connections and received messages to be handled over a greater number of threads. However much of the behaviour in this pool may involve locking the clients or parts of the topic tree. This can cause lock contention that delays processing.

Due to the underlying implementation of Java NIO sockets a high rate of threads being added/removed from the incoming thread pool will result in the allocation of off-heap byte buffers. In extreme cases this can result in an out of memory exception being thrown as the server runs out of off heap allocation space. For clients deploying on JDK 1.6 the problem is particularly severe as JDK 6 is much slower to reclaim unused off-heap buffers. As a result it is recommended that clients set the core and the max of the inbound thread pool to be the same to avoid thread pool churn.

Client Reconnection

Configuring the Client Reconnection Feature

Normally when a Client application loses its connection to the server, perhaps due to some communications error, then the only option is for it to connect again and then re-establish the state of the topics to which it was subscribed. There is however, the facility for Clients to be able to 'reconnect' a lost connection without losing its Topic state or messages that were queued for it whilst it was not connected.

This may be particularly useful for mobile Clients where connections are less reliable.

Server Configuration

How to Configure Client Reconnection at the Server

To allow for clients to reconnect then connectors must be configured to 'keep alive' Client sessions which have been disconnected due to an IO error. To do this a keep-alive time must be specified for the connector.

When a client is unexpectedly disconnected due to an IO error then instead of immediately closing the Client session it will be retained for up to the amount of time specified before it is finally closed. All of the Topic subscription state for the Client is maintained during this disconnected period and Messages for the Client will continue to be queued.

If the Client then reconnects before the keep alive period has expired then the sending of Messages to the Client will resume from the point when the failure occurred. Messages that were in transit at the time of disconnection could be lost.

The only way to guarantee the delivery of messages on reconnection is for the publisher to mark the messages as 'requiring acknowledgement' as as any such messages that have been sent but not acknowledged on reconnection will be requeued for the client. The delivery of 'acked' messages is therefore guaranteed, however because an ack from the client may have been lost during the disconnection there is the possibility that a message could be delivered to the client twice in this scenario.

It is important when using acknowledged messages to guarantee delivery after reconnect that the ack timeout set for messages is sufficiently long to allow for the time that a message may be queued for a client plus the keep alive time configured for reconnection.

If an ack timeout expires before a message is even dequeued for a client then the non acknowledgement will have been notified and the message will not be sent to the client.

Message Queue Management

Managing Message Queues when using Client Reconnection

When a client session is being kept alive messages for the client will continue queueing for the client until the keep alive period expires or the client reconnects. This will put an unusual load on the client queue and therefore the facility exists to adjust the maximum client queue depth for the period of disconnection.

Copyright 2013 Push Technology

This is done by specifying a keep alive queue depth which is greater than the normal maximum queue depth. When disconnected the queue can expand to the higher value and when reconnection occurs and the queue starts to drain then when the queue size goes down to a value of 80% of its previous limit then the maximum queue depth will revert to the normal value.

The keep alive queue depth only has an effect if it represents a value higher than the normal maximum queue depth.

Client Reconnection

Configuring reconnection of clients

Not all Clients support reconnection but those that do will have a 'reconnect' method which they may call on notification of a lost connection.

If the reconnection succeeds then the Client will be subscribed to all of the same Topics as before and will start to receive Messages again, including all of those queued whilst the Client was disconnected.

Messages in transit at the time of disconnection could be lost, however any message marked as requiring acknowledgement and sent by the server that was not acknowledged by the client will be retransmitted on reconnect. The delivery of acked messages from Client to server is therefore guaranteed on reconnect although there is the possibility that the client could receive a message it had acked before the connection again after reconnection if the ack had never reached the server.

A reconnection may not succeed, either because keep alive is not specified on the connector that the Client has connected to, or the keep alive time period has expired. In this case a normal new connection will be established with the same Topic set as was specified on the original connection.

How to test reconnection in my environment?

To simulate a comunication error, we use a proxy between the client and the server.

- Start Diffusion™. (By default it uses port 8080)
- Set the Proxy to listen on a different port (e.g 9090) and redirect the connection to 8080
- Connect the Client through the Proxy on port 9090.
- Kill the Proxy.
- Start the Proxy.
- Reconnect the Client.

Copyright 2013 Push Technology

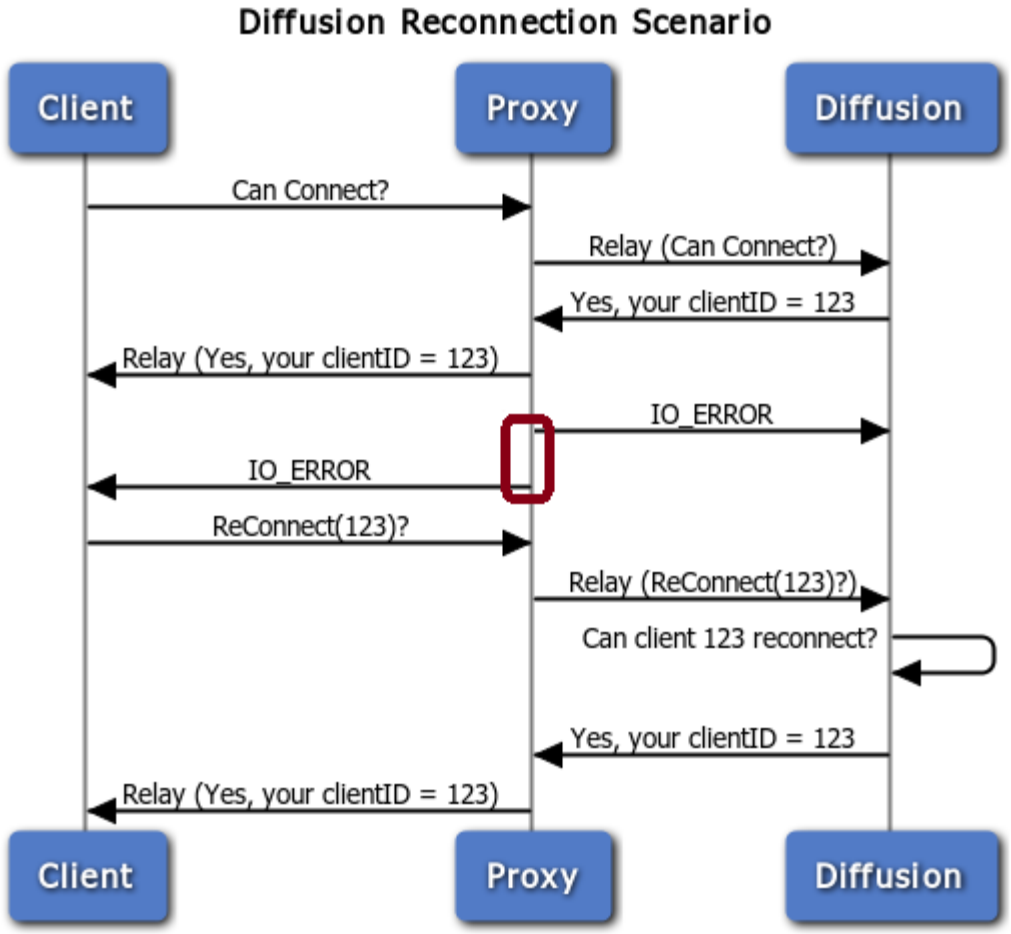


Figure 60: Reconnection Scenario

Note: The Proxy behaviours are different depending on the operating system and the TCP/IP stack configuration - this is beyond the scope of this document.

Common errors

1. From the client, request a connection close and then call reconnect.

   After a close request, the client cannot reconnect. In this case, the client will establish a new connection with a different client ID assigned by Diffusion™.
2. Unplug the network wire from the computer where the client is running

   This will not throw an IO_ERROR in the other end of the connection.

Client Failover

Configuring Clients to Fail Over to another Server

Client failover is when a client loses its connection to a server and attempts to connect to a different one. The client is provided with a list of servers. If a client loses its connection to a server it can automatically attempt to connect to the next server in the list. If it fails to connect or loses its connection to that server it will try the next server on the list. This is refered to as autofailover. Generally the list of servers to connect to must be provided before attempting to make the connection. How the list of servers is provided differs between Client APIs and the JavaScript client does not support autofailover but it can be implemented using the callback methods.

Copyright 2013 Push Technology

Autofailover

Using automatic Failover

If a client has an established connection that it loses autofailover will attempt to open a new connection in the next connection in the list. This is not compatible with reconnection because reconnection attempts to preserve the state of the client (the client ID and the subscribed topics). As the new server has no knowledge of the client it is unable to preserve this state. Autofailover is different to cascading which attempts to connect to the same server using different protocols before a connection has been opened. Autofailover must be enabled and a list of servers to connect to provided.

Java Failover

Configuring Failover in Java

In Java the ConnectionDetails object supports a collection of ServerDetails objects. The server details are use the control failover between servers. The ConnectionDetails factory methods provide serveral options for creating ConnectionDetails with multiple ServerDetails objects. A collection of ServerDetails object can be passed as a parameter, a varargs method supports ServerDetails and another varargs method supports String URLs which ServerDetails objects will be constructed from. After construction the ServerDetails objects used by the ConnectionDetails can be altered by calling the `setServerDetails(Collection<ServerDetails>)` method.

This code:

```
ConnectionDetails details =
    ConnectionFactory.createConnectionDetails(
        "dpt://192.168.0.1:8080",
        "dpt://192.168.0.2:8080");
details.setAutoFailover(true);
ExternalClientConnection client = new ExternalClientConnection(listener,
 details);
client.connect();
```

supports autofailover from the server with the IP address 192.168.0.1 to the server 192.168.0.2. If the client loses connection to 192.168.0.1 it will try to connect to 192.168.0.2. It uses the varargs method to create a ConnectionDetails object with mutliple ServerDetails objects constructed from String URLs. For further information refer to the Java API documentation for ConnectionDetails and ServerDetails.

JavaScript Failover

Using Failover in Javascript

The JavaScript client does not support autofailover. Support for failover is limited. If the connection attempt fails DiffusionClient.connect(DiffusionClientConnectionDetails) can be called with a different object. You must provide the logic to do this on connection failure.

ActionScript Failover

Using Failover in Actionscript

In ActionScript the ConnectionDetails object supports an array of ServerDetails objects. The server details are use the control failover between servers. The ConnectionDetails constructor has a mandatory ServerDetails object. After construction additional ServerDetails objects used by the ConnectionDetails can be altered by calling the `addServerDetails(ServerDetails)` method and the `setServerDetailsArray(Array)` method.

This code:

```
var server0:ServerDetails = new ServerDetails("dpt://192.168.0.1:8080");
var server1:ServerDetails = new ServerDetails("dpt://192.168.0.2:8080");
var details:ConnectionDetails = new ConnectionDetails(server0);
details.addServerDetails(server1);
details.setAutoFailover(true);
```

Copyright 2013 Push Technology

```
var client:DiffusionClient = new DiffusionClient();
client.connect(details);
```

supports autofailover from the server with the IP address 192.168.0.1 to the server 192.168.0.2. If the client loses connection to 192.168.0.1 it will try to connect to 192.168.0.2. It uses the `addServerDetails(ServerDetails)` method to add a single additional server to connect to. For further information refer to the *Flex API docuementation* for ConnectionDetails and ServerDetails.

Message Conflation

What is Conflation?

'Conflation' of Messages typically refers to the facility to treat two Messages as being essentially the same and thus avoiding sending duplicate information to Clients. This would involve removing an existing message from the outbound Client queue and replacing it with a newer 'equivalent' Message either in-situ (at the position of the old Message) or at the end of the Client queue (thus preserving absolute Message order.

For example, if a Client application is being updated with the price of a product, several price changes may occur in rapid succession and be queued for the Client. However, the Client is really only interested in the latest price, not any historic ones. Conflation is the ability to remove such superfluous Messages before they are sent to the Client thus reducing data transmission and Client processing.

Diffusion™ also supports 'Structural Conflation' (otherwise know as 'Merging conflation') which is where the content of the existing queued Message is merged with the content of the new Message to produce a new 'merged' Message. This enables all of the data from the two Messages to be sent to the Client but as a single Message.

Normally a Message is deemed to be 'equivalent' to another Message by virtue of the fact it belongs to the same Topic. However, it is also possible to only match Mesages of the same Topic for conflation according to some other criteria based upon the Message content.

How conflation is to take place for Topics is configured in terms of 'Conflation Policies' which define how Messages are matched, whether replacement is done in-situ or by appending, and optionally how to merge the two Messages.

Conflation is an optional feature that may be applied to all Clients, Clients connecting via a specific Connector or can be applied programatically on a Client by Client basis.

Conflation Overview

An overview of what Diffusion Conflation is

Technical Overview

Conflation is implemented as a lock-free, wait free algorithm and can therefore be scaled to meet demands of large numbers of concurrently subscribed clients. Slow Clients, such as those on embedded devices or available over unreliable, low bandwidth connections receive less frequent updates than local area clients, or low latency high bandwidth clients.

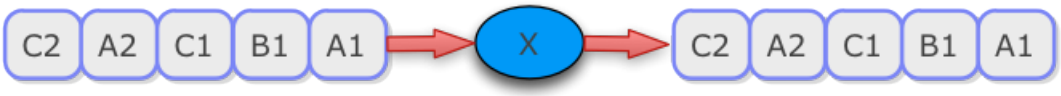
This technique is possible because Diffusion virtualizes client queues on the server, enabing monitoring of the ability of individual clients to sustain distribution levels and detect when they get saturated. Diffusion also caches data in memory and can introspect the transmitted data structures to distribute only the subset of data that has changed. Structural conflation builds on this to optimize data distribution by minimizing stale or soon to be replaced data from high frequency recoverable data streams.

Diffusion can be configured without conflation and with message acknowledgement for guaranteed delivery and detection of lost messages where appropriate.

Since most high frequency data streams are appropriate for structural conflation the technique can offer significnt resource savings whilst offering more current, consistent data delivery to clients.

Copyright 2013 Push Technology

Operation without Conflation

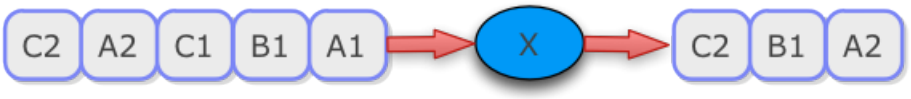


With no conflation, a stream of messages (on the left hand side) to a Client (right hand side) will be delivered to clients in the order that Publishers publish (or send) them. In the above case two messages for Topic "A", one message for Topic "B" and two messages for Topic "C" are ready to send to the Client.

This is a scenario common to all messaging platforms.

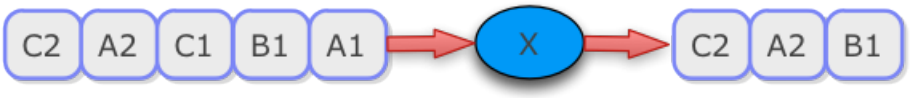
Simple Conflation

In its simplest form Conflation can replace a message that is yet to be delivered for a given Topic with a newer vaue for the same Topic but preserving the original message order. In other words, using the slot of the earlier value but the value of the latest change.



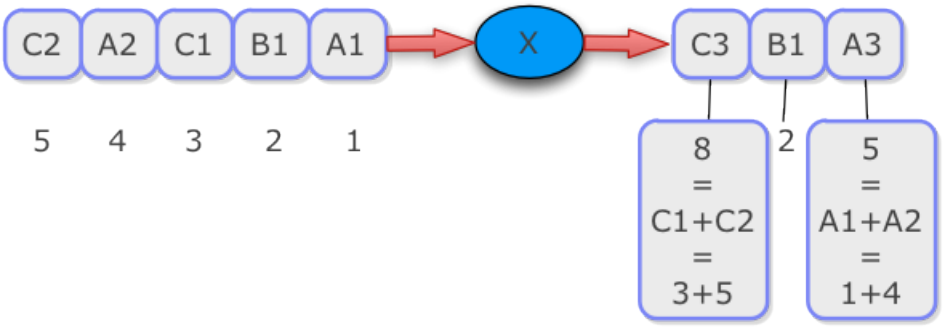
In the above example, although five messages were ready to send, only 3 messages were sent. This saves bandwidth and ensures clients receive current data only.

Alternatively the previous message for a Topic can be removed from its slot and the newer value queued at the end of the client queue. This option may be used if it is important that values are delivered in time order, however this should be used with care as there is the possibility that messages for a Topic could keep going to the end of the queue and a value not be delivered for the Topic. When this mode is used the resulting message order is different as shown below:-



Merging Conflation

Merging Conflation allows a user defined operation to be plugged into Diffusion so that rather than refreshing stale data with fresh data, a computation can be performed to merge, aggregate, reverse or combine the effects of multiple changes into a single consistent and current notification to the Client.



In the above example the operation is the summation of primitive data types such as numeric data. The user provides the merge algorithm, that is the summing of the values of two successive messages. Diffusion then sends the single resulting message rather than the individual messages that were combined. The messages A3 and C3 are new messages generated from the merging process.

This is suitable for any scenario where the result is required but individual components that combine to form the result are not.

Copyright 2013 Push Technology

The above example shows merge and replace. It is also possible to merge and append in a similar way as described for simple conflation above.

Various options are available to the user written merger so that instead of returning a merged message it could indicate that either of the input messages should be queued or that the no conflation option should be chosen.

Selection of Messages for Conflation

In the above examples it is assumed that when a new message is to be queued for a Client then it will replace or merge with the last message queued for the message Topic. This is the default behaviour. When operating conflation in this way then there should only ever be one message per conflated Topic awaiting delivery to the Client.

It is also possible to specify a user defined 'matcher' that will be used to determine the message that is to be replaced or merged with. This can be used to inspect the content of the messages queued for a Topic in order to select which to conflate. When operating conflation in this way it is likely there could be more than one message per conflated Topic awaiting delivery to the Client.

Considerations when using Conflation

- Conflation favours currency and reduces (but may not entirely eliminate) the delivery of stale data.
- When using conflation that alters message order it is assumed that there are no relationships or dependencies across the Topics concerned. That is to say a Topic is not temporally or causally related to any other Topic.
- Consistency is tunable and a function of merge behaviour. So for primitive data types a merge should be fully consistent (e.g. merge summation) but for complex data types (e.g. a order book) this may be more complicated.
- Composing conflation with *throttling* maximally reduces bandwidth whilst enabling current and consistent data delivery to clients. Configuration of throttling along with conflation must be done with care.
- There are cases where conflation of any kind should absolutely not be used. For example when individual messages carry forensic storage or audit trail requirements.

Conflation Business Value

What is the business value of using conflation?

Currency

There are many scenarios in real-time data distribution where data being communicated need not only be current, but must always be consistent. Structural Conflation maximises for concurrency whilst ensuring consistent delivery.

Both simple and merging conflation maximise for concurrency through avoiding distributing soon to be stale data. The higher the rate of change, the higher the value extracted. On other words, where clients or servers are running near to saturation based on available connectivity Diffusion can automatically adapt to this load by minimizing the data distributed.

In addition to the load-adaptive nature of confation the effect is fair for clients connected to the same Topic. The frequency of distributed changes is evenly amortized across clients.

Consistency

Structural conflation synthesizes Complex Event Processing techniques in a highly efficient (lock-free wait-free concurrency) form inside the server. The specific knowledge of data structures and the semantic concenrns for distributing data for a given topic in a given system allows consistent views of the data to be delivered in a way that is not possible with messaging technologies that treat messages as opaque.

How Does Conflation Work?

An outline of how Diffusion Conflation works

When Conflation is enabled for a Client then every time a new Message is enqueued for the Client then it will be considered for Conflation.

Copyright 2013 Push Technology

The first step is to check if there is a Conflation Policy that has been mapped to the Topic that the new Message is for. If there is no Policy then the Message is added to the end of the queue as normal.

If there is a Policy for the new Message then the 'matching' criteria of the Policy is used to scan the queue (from back to front) for an existing Message that 'matches' the one being queued. If no match is found then the new Message is queued at the end of the queue as normal.

Note that fast caches and lookup mechanisms are used to find Policies and Messages in the queue. The whole Client queue is not scanned when looking for a match, only Messages for the same Topic. And if the default matching (by Topic) is used then there is not even any need for a comparison with the existing Messages. This means that the Conflation mechanism is as efficient as possible.

Having found a matching Message in the queue, the Conflation Policy will then indicate whether the Message to be queued should be the new Message or a Message produced by merging the content of the current and new Messages. The policy also indicates whether the Message to queue (whether that be the new Message or a merge result) should replace the existing Message or whether the existing Message should be removed and the new Message added to the end of the queue.

Conflation occurs on a Client by Client basis in the Multiplexer thread. Results of merge actions are cached to ensure that merges of identical Message pairs are not repeated for different Clients.

Note the following important points about conflation:-

- Topic Load Messages are not conflated - only delta Messages.
- Only Normal priority Messages are conflated, not High or Low.
- Fragmented Messages are not conflated.
- Messages requiring Acknowledgement are not conflated.

Configuring Conflation

How to Configure Conflation

Conflation is configured by defining one or more Conflation Policies which describe how the conflation is to be done and then mapping Topics to those Policies.

Conflation Policies may be configured in the '`conflation`' section of the *`etc/Server.xml`* properties file or may be configured programatically using `ConflationConfig`.

Conflation Policies

What is defined in a Conflation Policy

One or more Conflation Policies may be configured, each defining different conflation mechanisms.

Conflation Policies may be configured in the `conflation` section of *`etc/Server.xml`* using `conflation-policy` elements.

Conflation Policies may also be configured programmatically using the various `addPolicy` methods on `ConflationConfig`. Such Policies may also be added dynamically after the Diffusion™ Server has started.

Conflation Policies comprise the following:-

Table 58: Conflation Policy Elements

Property	Description
name	A unique name by which the Policy is referred to.
mode	Indicating whether the new (or merged) Message is to replace the current Message in-situ or whether the current Message is to be removed and the new one appended to the end of the queue.
matcher	A Java class which matches two Messages and is used to locate an existing queued Message as a candidate for conflation.

Copyright 2013 Push Technology

Property	Description
	If no matcher is specified then default matching finds a Message that is of the same Topic.
merger	A Java class which performs the merge of two messages of the same Topic to produce a new Message containing the data from both Messages (or any resulting data required). If no merger is specified then no merging takes place and the current Message will be removed from the queue and the new Message will either replace it or be appended to the queue depending upon the mode. The merger can also indicate that either the current or new Message is to be used or even that no conflation should take place in this instance.

Having defined one or more Conflation policies then Topics may be mapped to them. This is done by specifying a Topic name or a Topic selection string (regex pattern) which maps to a particular Conflation Policy.

Conflation Policies may be added or removed at runtime and the removal of a Conflation Policy automatically removes any mappings to it.

Conflation Policy Mode

Modes of Conflation

The Conflation Policy Mode determines whether the new (or merged) Message is to replace the existing Message in the Client queue or be appended to the end of the Client queue.

Possible modes are:-

Table 59: Conflation Policy Modes

Mode	Definition
`REPLACE`	The new (or merged) Message will replace the existing Message at its current position in the Client queue.
`APPEND`	The current Message will be removed from the Client queue and the new (or merged) Message will be appended to the end of the queue.

If no mode is specified then `REPLACE` is assumed.

The mode is specified in the `mode` property of a `conflation-policy` section in *`etc/Server.xml`* .

When defining Conflation Policies programmatically the mode is specified when creating the Policy.

Message Matchers

How to use Message Matchers

A Message Matcher is used by a Conflation Policy when queueing a new Message for a Client that has conflation enabled for a Topic that has a Conflation Policy defined for it. The Message Matcher is used to locate the last Message queued for a Client that is a candidate for conflation.

If no Message Matcher is explicitly defined for a Conflation Policy then a default matcher is used which locates a Message of the same Topic.

A Message Matcher may be supplied if the matching is to be somehow dependent upon the content of the Messages.

To implement a Message Matcher you need to write a Java class that implements `com.pushtechnology.diffusion.api.conflation.MessageMatcher`. This has a single method called `matches` to which is passed the current Message in the queue being tested and the new Message to be queued. Note that the existing Message will always be of the same Topic as the new Message so there is no need to check that is the case.

Copyright 2013 Push Technology

An example of a `MessageMatcher` implementation is shown below:-

```
public class ExampleMessageMatcher implements MessageMatcher {
    @Override
    public boolean matches(TopicMessage currentMessage,TopicMessage
 newMessage) {
        return currentMessage.getHeader(0).equals(newMessage.getHeader(0));
    }
}
```

`MessageMatcher` implementations must be threadsafe and stateless. The same `MessageMatcher` instance may be supplied to more than one different Conflation Policy if so required.

Message Mergers

How to use Message Mergers

A Message Merger may be specified on a Conflation Policy if the action of the Policy is to merge the content of an existing queued Message with the new Message being queued. This technique can be used when Message data comprises more than one data item and it is desirable to reduce the number of Messages sent to the Client whilst preserving the data from all Messages.

If no Message Merger is specified for a Conflation Policy then the Policy will replace the current Message with the new.

To implement a Message Merger you need to write a Java class that implements `com.pushtechnology.diffusion.api.conflation.MessageMerger`. This has a single method called `merge` to which is passed the current Message in the queue and the new Message to be queued. Note that the existing Message will always be of the same Topic as the new Message.

The action of conflation will depend upon the Message that is returned from the merger method, as follows:-

Table 60: Action depending upon merge result

Returned Message	Action
A new message	It is assumed that the new Message represents a merging of the data of the two Messages input and so the returned Message will either eplace the current Message in the queue or the current Message will be removed and the retuned Message added to the end of the queue, depening upon the Policy mode.
The current Message	The current message is retained at its current queue position and the new Message is not queued.
The new Message	The new Message either replaces the current Message in the queue or the current message is removed and the new Message appended to the end of teh queue depending upon the Policy mode. This is effectively the same as the result would have been if there had been no merger.
Null	No conflation will occur. The current Message remains where it is in the queue and the new Message is appended to the end of the queue,

An example of a MessageMerger implementation is shown below:-

```
@Override
public TopicMessage merge(TopicMessage currentMessage,TopicMessage
 newMessage) throws APIException {
    TopicMessage result =
 Publishers.createDeltaMessage(currentMessage.getTopicName());
    Map<String,String> values = new LinkedHashMap<String,String>();
    while (currentMessage.hasRemaining()) {
        values.put(currentMessage.nextField(),currentMessage.nextField());
    }
```

Copyright 2013 Push Technology

```
    while (newMessage.hasRemaining()) {
        values.put(newMessage.nextField(),newMessage.nextField());
    }
    for (Entry<String,String> entry:values.entrySet()) {
        result.putFields(entry.getKey(),entry.getValue());
    }
    return result;
}
```

The above example merges two sets of key value pairs from the two messages with the new message values overwriting any duplicates in the current message.

`MessageMerger` implementations must be threadsafe and stateless. The same `MessageMerger` instance may be supplied to more than one different Conflation Policy if so required.

Default Conflation Policy

Using a default Conflation Policy

It is possible to specify a default Conflation Policy that will be used for any Topics that do not have explicit Policy mappings.

This should only be used if you wanted to apply conflation to all Topics when conflation is enabled for a Client.

This can be specified using the `default-conflation-policy` property in the `conflation` section of *etc/Server.xml*. Alternatively it can be set programatically at any time using the `setDefaultPolicy` method on `ConflationConfig`.

If no default Policy is set then conflation will not occur for Topics that have no explicit mappings even when conflation is enabled.

Mapping Topics to Policies

Mapping Topics to Conflation Policies

Having defined one or more Conflation Policies it is then possible to map Topics to the Conflation Policies that are to be used for them.

Either a full Topic name or a Topic Selector pattern (regex) may be used when mapping to a Conflation Policy.

As the use of Topic Selectors means it would be possible for more than one mapping to potentially apply to the same Topic then the last mapping defined that matches a specific Topic will be the one that is used for conflating Messages of that Topic.

A default conflation policy may be specified which would be selected to map to if no other mapping matches a Topic.

Messages for Topics that have no mappings (when there is no default policy) are not conflated, even if conflation is enabled for a Client.

Conflation mappings may be defined using `topic-conflation` elements within the `conflation` section of the *etc/Server.xml* property file.

Conflation mappings may also be set programmatically using the `setTopicPolicy` method of `ConflationConfig`. Mappings may be set at any time during the running of a server. Mappings may also be removed at any time using `unsetTopicPolicy`.

Note that removing a Conflation Policy at runtime will automatically remove any mappings to it.

Enabling Conflation

This describes how to turn conflation on and off

Having configured the policies that define how conflation is to be performed on a per Topic basis then conflation may then be turned on either for all Clients connecting via a specific Connector or for individual Clients.

Enabling conflation means that conflation will occur for any messages queued for clients that have Conflation Policies set for their Topic(s).

Enabling Conflation via XML Configuration

Copyright 2013 Push Technology

Conflation may be specified for a queue-definition in the *etc/Server.xml* configuration file by setting the 'conflates' property to 'true'. This queue definition may then be used wherever required, for example by Connectors that wish to have conflation enabled for all Clients.

Enabling Conflation for a Connector Programmatically

When programatically starting a Diffusion™ Server then conflation can be specified in a similar way to within XML by setting the 'conflates' property on a queue definition (`QueueConfig`).

Enabling Conflation for a Specific Client

Conflation can be turned on (or off) programmatically at any time for a connected Client. For example, you may choose to start conflating a Client's mesage queue as a result of the queue's upper threshold having been exceeded. To turn conflation on for a Client use the `setConflation` method on the `Client` interface.

Conflation Counts

Determining whether Conflation has occurred

To determine whether Conflation has occurred for a `Client` you can call the `Client.getConflationCount` method which returns the number of conflations that have occurred for the Client.

This also has the facility to reset the count when called.

Client Throttling

Limiting the Message Volume to Clients using the Throttling Feature.

'Throttling' is a method of ensuring that the Diffusion™ Server, will limit ("throttle") the volume of messages it transmits to a Client within a specified period of time. This may be used to limit bandwidth usage or to prevent more Messages being sent to a Client than it can cope with.

How Does Throttling Work?

How throttling works

Throttling is applied to a Client queue so that the number or volume of Messages sent to that Client is restricted. Diffusion™ will only dequeue a Message to send to a Client if that Client has not breached its Throttling limit.

Throttle types;

- Messages per Second (Only a specified number of messages are sent every second.)
- Bytes per Second (Only a specified number of bytes are sent every second)
- Message Interval (A single message is sent every n millseconds.)
- Buffer Interval (A full output buffer (or the equivalent of) is sent every n millseconds)

Copyright 2013 Push Technology

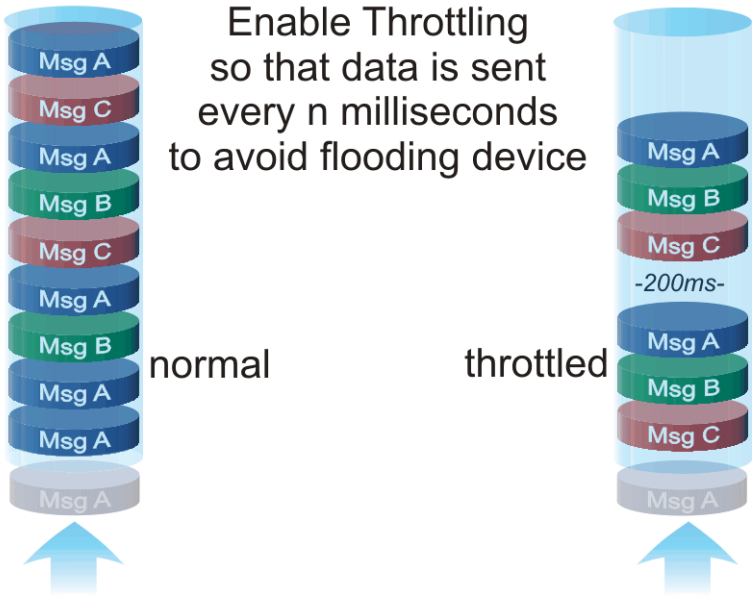


Figure 61: Normal and Throttled Client Queues

Enabling Throttling

How to enable Throttling

Throttling can only be enabled on a Client by Client basis from within the Publisher.

To throttle a Client.throttle method which allows you to specify the type of throttling and a limit. A ClientThrottler reference is returned.

If the throttle method is called for a Client that is already throttled then it has the effect of removing the old throttler and adding a new one.

Call the Client.removeThrottler method to stop throttling.

The Client.isThrottling method may be used to determine whether a Client is currently throttled and if it is then the getThrottler method may be used to determine the type of throttling and the current limit.

Memory Considerations

Memory Usage in the Java VM

Garbage Collection (Java HotSpot VM)

Garbage Collection Considerations

Typically you would not need to alter the default garbage collection for the Diffusion™ VM but depending upon the nature of the application then better performance and memory usage could be obtained by tuning the VM's garbage collection parameters. This is a complex area and the Oracle guides to garbage collection tuning should be consulted. However, some considerations are mentioned below:-

Generational vs Incremental Collection

The default garbage collection algorithm used is generational which means it views newer object references differently from older ones and assumes that what is to be collected is generally in the newer space.

This approach may be generally OK but in the case where Diffusion™ is maintaining message queues for clients then it is possible that messages remain in a queue for long enough to be moved out of the space eligible for regular collection. The overall effect of this might be that memory (heap) usage appears to grow indefinitely and a sudden peak of memory usage could result in an 'out of memory' error. This can be overcome by use of the 'incremental

Copyright 2013 Push Technology

garbage collector' which will do period sweeps of older objects. This may be done using the `-Xincgc` flag for the Diffusion™ Java VM.

Platform Specific Issues

To run Diffusion™ it may be necessary to increase the number of sockets and reduce timewait.

It may also be necessary to increase the number of open files that is allowed on UNIX/Linux systems

Socket Issues

To fix these problems, complete the follow steps based on platform.

Windows

Setting Values on Windows

Setting TCP timed wait

This parameter determines the length of time that a connection stays in the TIME_WAIT state when it is closed. When a connection is in the TIME_WAIT state, the socket pair cannot be reused. This is also known as the 2MSL state because the value should be twice the maximum segment lifetime on the network. See RFC 793 for further details.

Add TcpTimedWaitDelay registry values as a workaround. You can set these values through REGEDIT command.

Set TcpTimedWaitDelay to 30:

1. Click Start > Run.
2. In the available field, enter regedit.
3. Go to the key directory file: `HKEY_LOCAL_MACHINE/SYSTEM/CurrentControlSet/Services/Tcpip/Parameters/TcpTimedWaitDelay`. The value type is `REG_DWORD`.
4. Double-click TcpTimedWaitDelay.
5. Select Decimal.
6. Type 30 in the Value data field. The default value for this field is 0xF0 (240 decimal). The valid range is 30-300 (decimal).

Setting MaxUserPort

This parameter controls the maximum port number used when an application requests any available user port from the system. Normally, short-lived ports are allocated in the range from 1024 through 5000. Setting this parameter to a value outside of the valid range causes the nearest valid value to be used (5000 or 65534).

Add MaxUserPort registry values as a workaround. You can set these values through REGEDIT command.

Set the MaxUserPort to 65534

1. Click Start > Run.
2. In the available field, enter regedit.
3. Go to the key directory file: `HKEY_LOCAL_MACHINE/SYSTEM/CurrentControlSet/Services/Tcpip/Parameters/MaxUserPort`. The value type is `REG_DWORD`.
4. Double-click MaxUserPort.
5. Select Decimal.
6. Type 65534 in the Value data field. The default value for this field is 0x1388 (5000 decimal). The valid range is 5000 - 65534(decimal).

Linux

Configuring Sockets Values on Linux

Copyright 2013 Push Technology

Decrease the time wait before closing the sockets by entering:

```
# echo 3 > /proc/sys/net/ipv4/tcp_fin_timeout
```

Sometimes systems are now configured to prevent one from using a large number of ports, check the port range and modify if required.

```
# cat /proc/sys/net/ipv4/ip_local_port_range
```

This can be increased by issuing the following command

```
# echo "1025 65535" > /proc/sys/net/ipv4/ip_local_port_range
```

To have these new values take effect you may have to do (as root)

```
# /etc/rc.d/init.d/network restart
```

If you want these new values to survive across reboots you can at them to `/etc/sysctl.conf`.

```
# Allowed local port range
net.ipv4.ip_local_port_range = 1025 65535
# net.ipv4.tcp_tw_recycle = 1
net.ipv4.tcp_fin_timeout = 3
```

Solaris

Configuring Sockets Values on Solaris

The tuning changes noted below do not remain across reboots. Add the changes to a runtime commands file /etc/init.d/inetinit to make the changes permanent.

1. Enter the following commands as root user to make more sockets available:
   a) At the command prompt, enter: `/usr/sbin/ndd -set /dev/tcp tcp_conn_req_max_q 65534`
   b) At the command prompt, enter:
2. Decrease the time wait before closing the sockets by entering:
   a) `# /usr/sbin/ndd -set /dev/tcp tcp_time_wait_interval 60000`

AIX

Configuring Sockets Values on AIX

The tuning changes noted below do not remain across reboots. Add the following changes to the runtime commands file (`/etc/rc.net`) to make the changes permanent.

1. To make more sockets available, enter at the command prompt:
   a) /usr/sbin/no -o clean_partial_conns=1
   b) This setting will instruct the kernel to randomly remove half-open sockets from the q0 queue to make room for new sockets.
   c) The default range is sufficient, but you can change it using the no command. The following example sets the TCP ephemeral port range to 49152 through 65535:
   d) /usr/sbin/no -o tcp_ephemeral_low=49152
   e) /usr/sbin/no -o tcp_ephemeral_high=65535
2. To make more sockets available, enter at the command prompt:
   a) # no -o tcp_timewait=5

On AIX 4.3.2, the default TIME_WAIT seems to be set to seconds, which translates in AIX terms to tcp_timewait = 1, as each tcp_timewait is about 15 seconds. So you need to use the no command to set the tcp_timewait to 5 to make the TIME_WAIT just over 60 seconds (using a value of 4 will leave TIME_WAIT less than 60 seconds)

Increasing Number of Open Files

How to Increase the Number of Open Files on Unix Systems.

Copyright 2013 Push Technology

The `ulimit` command can increase the number of open files. This can be done as a global setting for the box by your network administrator. If this setting cannot be changed globally then the start script for diffusion will need modifying. Edit the startdiffusion.sh file and add the following at the start:

```
ulimit -n 8192
```

The value can be changed to any suitable value. This command can only be run by a user with the right permissions.

Publisher Design

Considerations when designing a Publisher

The following are some points that should be considered when designing and writing a Publisher:-

Data Modelling	The way that the data is fed to a Publisher and the way in which the state of the data is maintained within a Publisher is key to good performance. Message sizes should be kept to a minimum and this can be achieved using fine data granularity enabled by hierarchic Topics.
Caching	Messages should be cached wherever possible rather than building new ones every time one needs to be sent. This particularly applies to Topic Load messages which can be cached to send to every new Client that subscribes and only rebuilt when the data actually changes. The ideal place to keep such cached Messages is with a data object attached to the Topic (see Topic Data pattern).
String Handling	Building of Strings by concatenation is very inefficient in Java and should be kept to a minimum. When String content is used then Message caching can help to some degree and wherever possible Strings that need to be built should be cached.
Conditional Processing	Excessive use of conditional processing (Checking of Topic names etc) can be expensive. Use of the Topic Data pattern can significantly reduce the need for such processing when many Topics are in use.
Topic Naming	As every Message must carry the Topic name then long Topic Names can lead to a large amount of data traffic which can be disproportional to the data being carried. This may be a particular problem when using Hierarchical Topics. A solution is to use the Topic aliasing feature so that only short aliases of the Topic names are transmitted in Messages.
Concurrency	Concurrent programming means that access to data often needs to be synchronised but care must be taken not to synchronise more than is necessary as performance could be significantly affected.

Copyright 2013 Push Technology

Copyright 2013 Push Technology

Chapter 21

Diagnostics

Topics:

- *Logging*
- *Logging API*
- *Connection Counts*
- *Message Diagnostics*
- *Javascript Diagnostics*
- *Flex and Flash Diagnostics*
- *Windows Diagnostics*
- *Debugging a Publisher*
- *Log Messages*

Diffusion™ provides various mechanisms for logging information and diagnosing problems. The major Internet browsers also include useful diagnostic tools. All are discussed in this section.

Logging (Java)	Diffusion™ provides a comprehensive logging facility which may be used from within Java, whether in Publishers or applications using Java APIs.
Message Diagnostics (Java)	Analysing Message Content from within the Java APIs.
Javascript Diagnostics	Analysing Message Content and issues utilising browser tools.
Flex / Flash Diagnostics	Analysing issues within the Flash framework.
Windows Diagnostics	Analysing issues within the Windows APIs.
Debugging a Publisher	Being able to step through debug a Publisher using the Eclipse IDE

Logging

The Diffusion™ Server logs messages about its activities and any problems that occur. Log messages are written to the console (the process output, which may appear in a terminal window) and to a log file. The level of logging can be adjusted, from a moderate rate of messages under normal circumstances for system monitoring, to very detailed logging for problem diagnosis.

Configuring Logging

The log messages created by the Diffusion™ Server, and by publishers deployed to the server, are filtered by configuration in `etc/Logs.xml` . The levels of logging for the console and log file may be configured independantly. The name and location of the file can be specified as required; files can be configured to rotate daily, based on file size, or both; and the number of old logs to keep can be set.

Application code that uses the Diffusion™ API but is not deployed in a server environment (i.e. code that is not part of a publisher) can still use the logging facilities. In this case, the API may be used to reconfigure the *logging back-end*, and `etc/Logs.xml` configuration does not play a part.

Log Levels

The *log format* was standardised in Diffusion™ 4.6. The log levels used in the log output changed from a variant of *java.util.logging* values to *SLF4J* values. To maintain compatibility with existing Diffusion™ deployments, the *java.util.logging* values are still used in the configuration in this release.

The log levels, in severity order, are as follows:

Configured value	Messages that will be written to logs (SLF4j values)
`SEVERE`	`ERROR` messages only.
`WARNING`	`WARN` and `ERROR` messages.
`ADVICE`	`WARN` and `ERROR` messages. `INFO` messages logged using `LogWriter.advice()`
`INFO`	`INFO`, `WARN`, and `ERROR` messages.
`FINE`	`DEBUG`, `INFO`, `WARN`, and `ERROR` messages.
`FINEST`	`TRACE`, `DEBUG`, `INFO`, `WARN`, and `ERROR` messages.

Warning: Logging can use considerable CPU resources, and only significant log messages (`INFO` and above) should be enabled in a production environment. Performance will be degraded significantly when running at lower logging levels as more messages are produced, each requiring processing.

Log Format

From Diffusion™ 4.6, the log format is formally documented. This eases integration with *log monitoring tools*.

Each log line is made up of a number of fields. All of the fields except for the *Exception* are formatted on a single line, delimited by pipe (`|`) characters.

```
2013-07-17 15:21:57.355|INFO|main|PUSH-000165|Diffusion Server started.|
com.pushtechnology.diffusion.DiffusionController
```

If a log event has an exception, the exception message and stack trace will directly follow the message line.

```
2013-07-17 16:57:50.659|WARN|main|PUSH-0000258|Failed to load publisher
 'ConsolePublisher'.|
com.pushtechnology.diffusion.publisher.PublisherManagerInstance
```

Copyright 2013 Push Technology

```
com.pushtechnology.diffusion.DiffusionException: Class
 com.pushtechnology.diffusion.apps.console.publisher.ConsolePublisher for
 Publisher ConsolePublisher not found
 at ...PublisherManagerInstance.createInternalPublisher(Publisher...)
 at ...PublisherManagerInstance.loadPublisher(PublisherManagerInstance.java:108)
 at ...PublisherManagerInstance.loadPublishers(PublisherManagerInstance.java:141)
 at ...PublisherManager.loadPublishers(PublisherManager.java:51)
 at ...DiffusionController.startImpl(DiffusionController.java:220)
 at ...DiffusionController.start(DiffusionController.java:126)
 at ...api.server.DiffusionServer.start(DiffusionServer.java:105)
 at ...api.server.DiffusionServer.main(DiffusionServer.java:140)
 at ...Diffusion.main(Diffusion.java:20)
```

The meaning of each field is described in the following table.

Field	Optional or Mandatory	Format/values stable between releases	Description
Time stamp	Mandatory	Yes	The time and date the log event occurred. Asynchronous logging is enabled by default. The server may log a message in a different thread to the one that produced the log event, and at a slightly later time. Consequently, log lines may not be logged in exact time stamp order. The time stamp is displayed using the timezone configured for the Java Virtual Machine (JVM) running the server. The date format can be changed in the *`Server.xml`* configuration file.
Severity	Mandatory	Yes	The log severity, using the SLF4J levels: `ERROR`, `WARN`, `INFO`, `DEBUG`, `TRACE`.
Thread	Mandatory	No	The name of the Java thread that logged the event.
Code	Optional	Yes	A code identifying the message type. All *documented messages* have a code. Example: `PUSH-000123`. These messages can be logged at any level, but are typically `INFO` or above.
Message	Mandatory	No	A natural language description of the event.
Logger name	Mandatory	No	The logger name. Usually the fully qualified name of the Java class that produced the event.
Exception	Optional	No	If the log event has an associated Java `Throwable`, the exception message and stack trace will directly follow the message line.

Optional fields will be empty if the log event does not have the information.

The third column indicates whether fields are stable between releases. Where possible, Push Technology will not change the format or values of these fields so they may be relied on for automated log monitoring. The fields not marked as stable are more likely to change between releases, including patch releases.

Log Headers

In this release, Diffusion™ log files start with a special header.

Copyright 2013 Push Technology

```
2013-07-08 09:58:12.095 : Starting log for Diffusion 4.5.4_01 (Server)
 (2013-06-07T01:02:19)
```

This header shall be replaced with a standard log message in a future release, and is not to be depended on.

Logging API

The Diffusion™ Java API includes a logging facility that can be used by applications to log information.

Prefer SLF4J

From Diffusion™ 4.6, the product implementation has switched to the *SLF4J* logging API.

In this release, the `LogWriter` API described below is still supported for application use. Push Technology recommends that new applications use SLF4J to log events instead of `LogWriter`.

Whether SLF4J or `LogWriter` is used, log events are processed by a shared *logging back-end* and are written to the same destinations.

The LogWriter API (deprecated)

Diffusion™ provides its own `LogWriter` interface that provides similar functionality to the JDK's `java.util.logging.Logger`. Push Technology recommends that new applications use SLF4J to log events instead of `LogWriter`.

Every publisher has its own `LogWriter` which can be obtained using the `Publisher.getLogger()` method. Logging performed from within a publisher may be directed to the console and/or files according to the *configuration settings*.

A `LogWriter` may also be obtained using the `Logs.getLogger()` method. This can be used in non-publisher APIs, and when not running in a server environment. By default, logging outside of the server will write messages to `System.err` and be filtered at `INFO` level. This can be changed by *configuring a new handler*.

`LogWriter` has a method for each level of logging (`SEVERE`, `WARNING` etc). The method you use determines the log level at which the message will be logged. For example, if you use the `fine` method then the message will only be logged if `FINE` logging (or lower) is enabled in the configuration.

Within a publisher, logging could be done as follows.

```
protected void publisherStarted() throws APIException {
    getLogger().info("Publisher "+this+" has started");
}
```

In contrast, from within within an application class running outside the server environment, logging could be done as follows.

```
Logs.info("A Message To Display");
```

`LogWriter` has methods to check whether a particular level of logging is enabled. This can be used to improve performance, avoiding the need to format messages that are never going to be used. For example:-

```
LogWriter logger = getLogger();
//..
if (logger.isFineLogging()) {
    logger.fine("Publisher " + this + " has generated some data");
```

Copyright 2013 Push Technology

```
}
```

It is recommended that the above technique is used in all but the highest levels of logging (i.e. those that probably are always enabled).

`LogWriter` has versions of the level logging methods which also take a `Throwable` parameter. This allows an associated exception to be logged along with the message. A full stack trace of any exception specified will be logged.

Mapping of LogWriter log levels

As described in *Configuring Logging* on page 422, the log levels were changed in Diffusion™ 4.6. The following table describes the mapping of the `LogWriter` methods to the level used in the log files.

`LogWriter` method	**Level used in log file**
`severe()`	`ERROR`
`warning()`	`WARN`
`advice()`	`INFO`
`info()`	`INFO`
`fine()`	`DEBUG`
`finest()`	`TRACE`

The logging back-end

SLF4J and `LogWriter` are both APIs that separate the code used to log a message from the logging implementation. The work of formatting and writing out messages is done by the logging *back-end*.

In this release, the Diffusion™ logging back-end is based on *java.util.logging*. Push Technology is considering changing the back-end implementation in a future major release. The information in this section is subject to change.

In a Diffusion™ server environment there are two log handlers - implementations of `java.util.logging.Handler`. One handler is responsible for writing messages to the console and the other for writing messages a file. In a non-server environment, there is a single default handler that logs to `System.err`.

In either case, additional handlers can be added to process log messages in any way that is required. For example, you might configure a logger that sends messages directly to a UNIX `syslog` process.

A handler may be added using the `LogWriter.setHandler()` method. You can add a handler that will process messages from a single publisher by adding it to the `LogWriter` returned by its `getLogger()` method. Alternatively you can add a handler to the root `LogWriter` obtained from `Logs.getLogger()` and it will handle messages for all loggers, including those used by the Diffusion™ software.

In a server environment, additional handlers do not replace the default handlers. In a non-server environment, the first handler added to the root `LogWriter` replaces the default handler.

Connection Counts

The Diffusion™ Server produces connection summaries.

At one minute past midnight Diffusion creates an entry in the file `logs/ConnectionCount`, and resets the counter.

The value in the third column is the total number of client connections that have been made that day.

The value in the fourth column is the maximum number of concurrent connections that have been active that day.

Copyright 2013 Push Technology

The scheduling of this reset defaults to once per day, but it may be controlled by supplying the `<reset-frequency>` element in *etc/Statistics.xml* .

An example is shown here:

```
2013-03-27 00:01:40     4.4.0_05     128     31
2013-03-28 00:01:48     4.4.0_05     139     28
2013-03-29 00:01:56     4.4.0_05     177     28
2013-03-30 00:01:05     4.4.0_05     118     41
2013-03-31 00:01:13     4.4.0_05     196     30
2013-04-01 00:01:22     4.4.0_05     207     36
2013-04-02 00:01:31     4.4.0_05     188     19
2013-04-03 00:01:41     4.4.0_05     244     44
2013-04-04 00:01:41     4.4.0_05     188     26
2013-04-05 00:01:41     4.4.0_05     195     39
```

Message Diagnostics

Within the Java API the `Message.toString()` method will always produce a String representation of the Message content including headers. This is very useful for diagnostic purposes but should be used with caution as there is a significant overhead to converting a Message to String format.

A Message will be rendered in String form as follows:-

`(L=` *n* `,T=` *n* `,E=` *n* `)[` *headers* `]` *data*

`L=` *n* shows the Message length, in bytes, including headers.

`T=` *n* shows the Message Type (see Protocol for possible values).

`E=` *n* shows the Message encoding (see ProtocolProtocol for possible values).

[*headers*] shows any Message headers. The *headers* String shows the headers separated by `<FD>` and terminated by `<RD>` (symbolising Field and Record delimiters). Fixed headers appear first followed by any user headers. If the Message has no headers then this part will not be present.

data shows a character representation of the data content of the Message. `<FD>` and `<RD>` are used to indicate Record and Field delimiters, if present. If the Message is encoded then the decoded data is displayed. If the Message has no data content then this will not be present.

It should be noted that the the Message length shown will not necessarily correspond to what is shown. For example, the length of a Message with compressed encoding could be significantly less than the length of the data displayed. It must be borne in mind that the part within `()` is always fixed at 6 bytes, the part within `[]` is UTF-8 encoded characters and the *data* may also be character or byte encoded which may affect its length within the Message.

The following example of code is followed by the actual Message representation that comes from the `Message.toString` method:-

```
message.setHeaders("H1","H2");
message.putRecord("A","B","C");
message.putRecord("D","E","F");
System.out.println(message);
```

.. would be displayed as:

```
(L=31,T=21,E=0)[MyTopic<FD>H1<FD>H2<RD>]A<FD>B<FD>C<RD>D<FD>E<FD>F
```

Copyright 2013 Push Technology

Note that there is no Field delimiter at the end as the Field delimiter is a separator and not a terminator. If there was a Field delimiter at the end it would imply an empty (zero length) Field at the end of the data. The same applies to Record delimiters.

Javascript Diagnostics

In-Browser Javascript Diagnostics

There are currently more than 30 Web Browsers in use, but for the purposes of this manual we shall limit our discussion to five of the more widely used browsers, namely Firefox, Chrome, Internet Explorer, Opera and Safari. The following instuctions describe how to enable and make use of the browser's consoles.

As Javascript is a typeless language, it can very difficult to debug. DiffusionClient surrounds both onMessage and topic listener function invocations with a try..catch block. This may mean that if there is an error in the functions that accept WebClientMessage, this may silently fail and not produce the expected results. In order to trap this, there is a DiffusionClient.trace(message) function. The trace function will create a message which can be seen in the browser's logging console. Runtime errors are also displayed within the console.

To allow debugging of the client, the following entry needs to be included;

```
<script type="text/javascript"> function startDiffusion()
 { DiffusionClient.setDebugging(true); } </script>
```

Mozilla Firefox

Firebug integrates with Firefox allowing you to debug and monitor JavaScript, and edit HTML and CSS. Firebug requires Firefox 3.5 or higher and can be downloaded from; *https://addons.mozilla.org/en-US/firefox/addon/1843*

To start debugging, click on the Firebug Icon

in the bottom right corner of the screen.

Ensure that logging is enabled in the drop-down menus for 'Console' and 'Net'

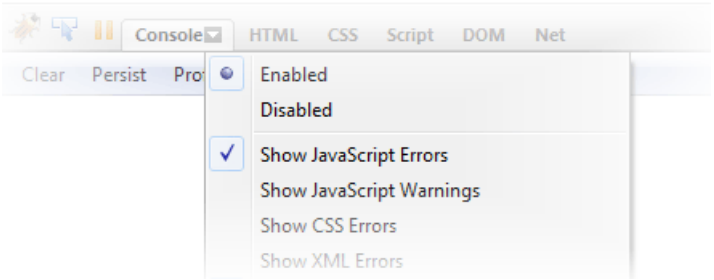


Figure 62: Firefox Console

Google Chrome

Chrome's console can be found in the 'Developer tools' section, as below -

Copyright 2013 Push Technology

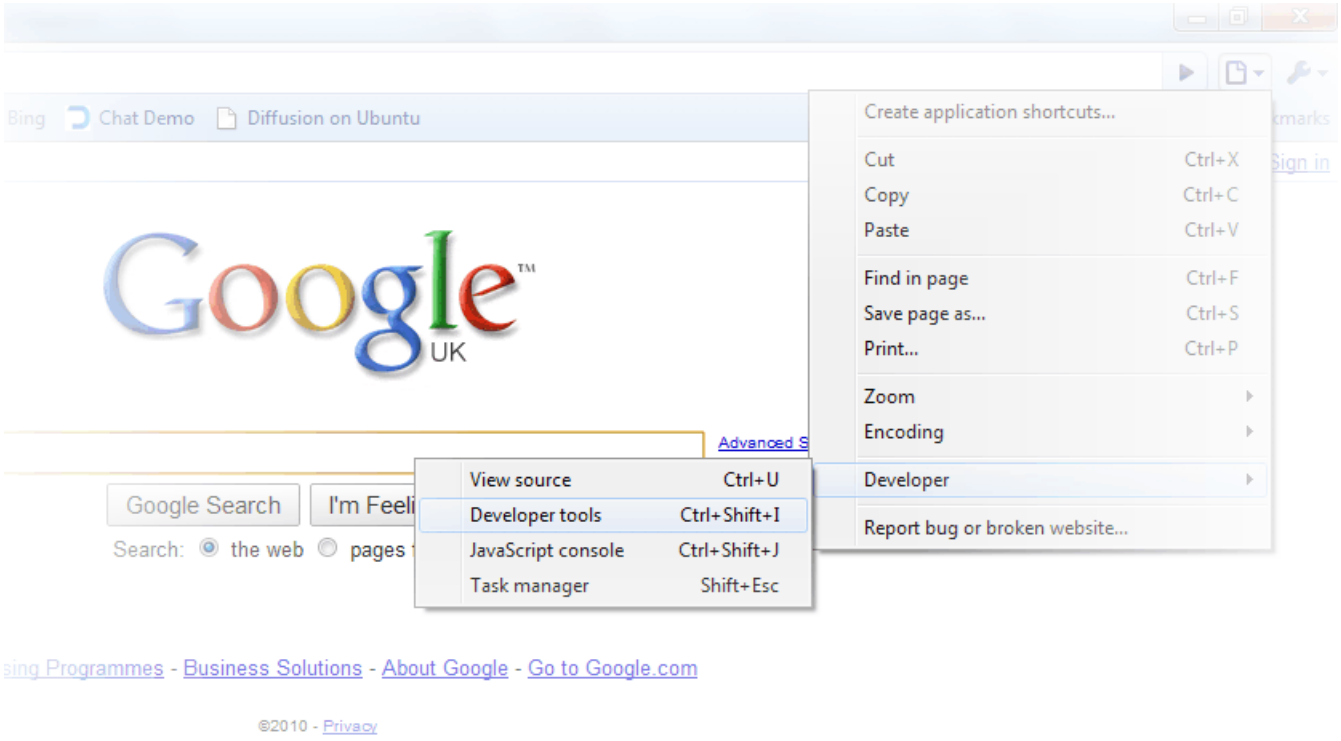
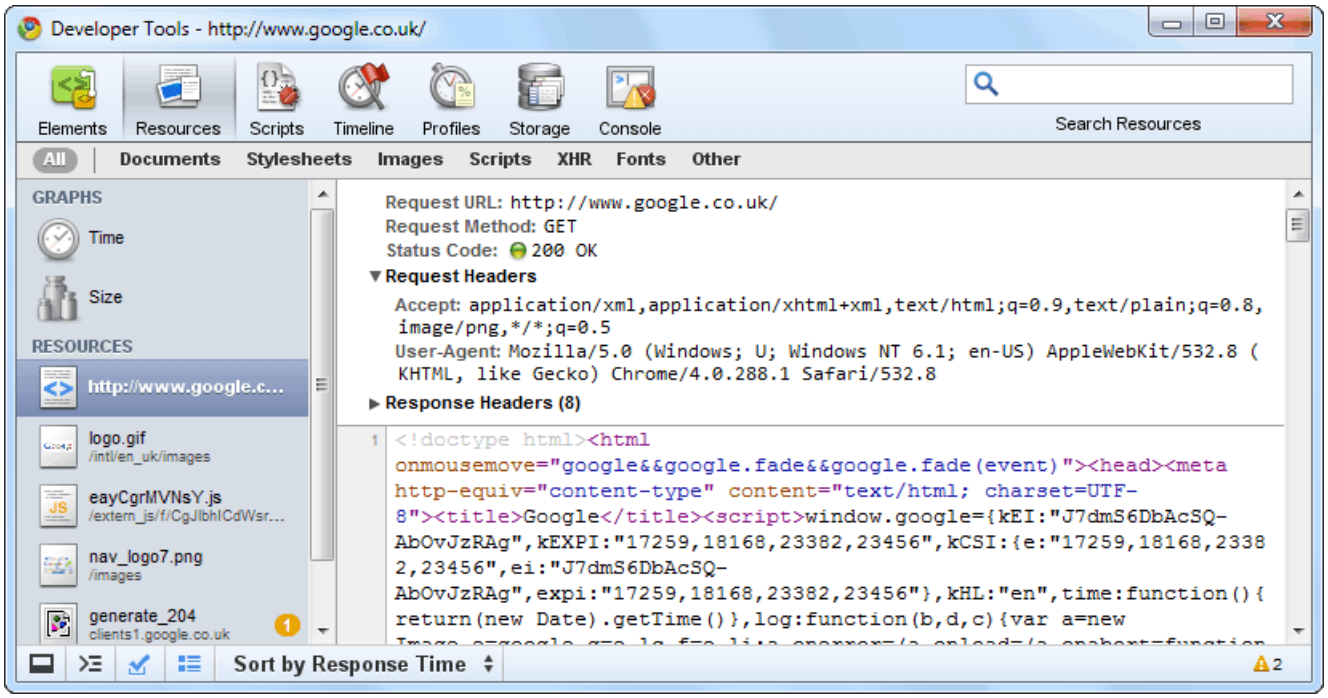


Figure 63: Chrome's console

- which will bring up the console. At first use you will need to click 'enable' when using the 'resource' tab.

t

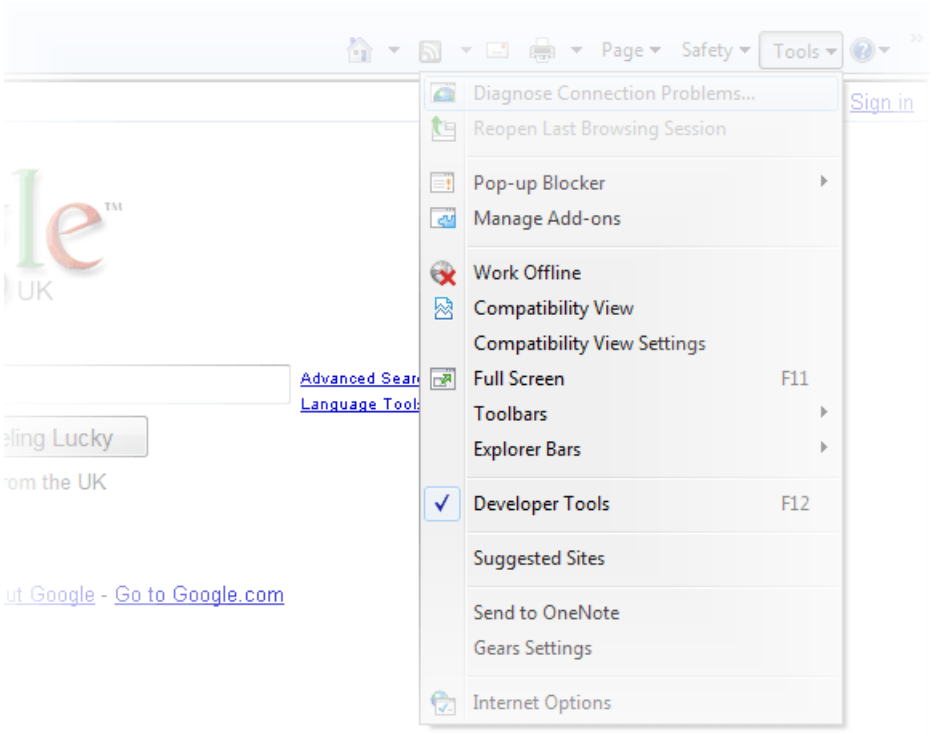


Internet Explorer

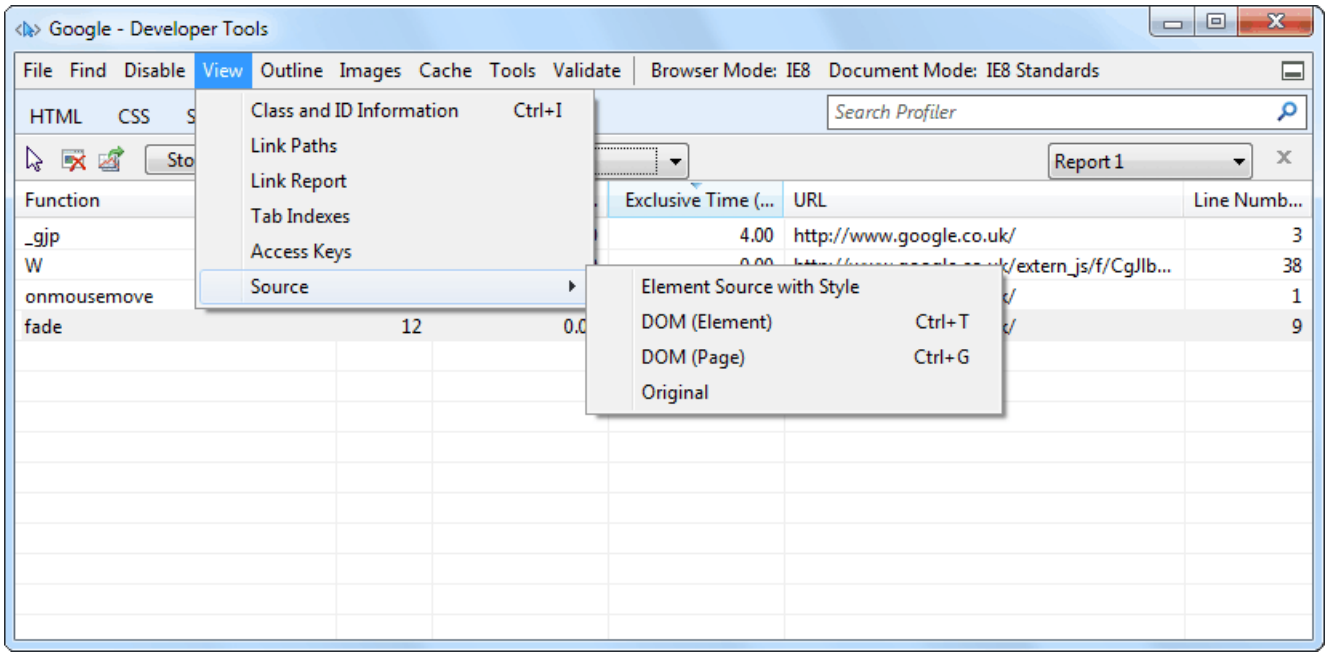
Copyright 2013 Push Technology

IE8 has a console which is accessed as follows -

Figure 64: Internet Explorer's console



- this will bring up the console. In some cases when attempting to start Developer tools, the window fails to appear when it is outside the viewport. This can be remedied by pressing 'alt' and 'space' then choosing 'restore'.



Copyright 2013 Push Technology

Opera

Opera's console is similar to that of Firefox, and is accessed thus -

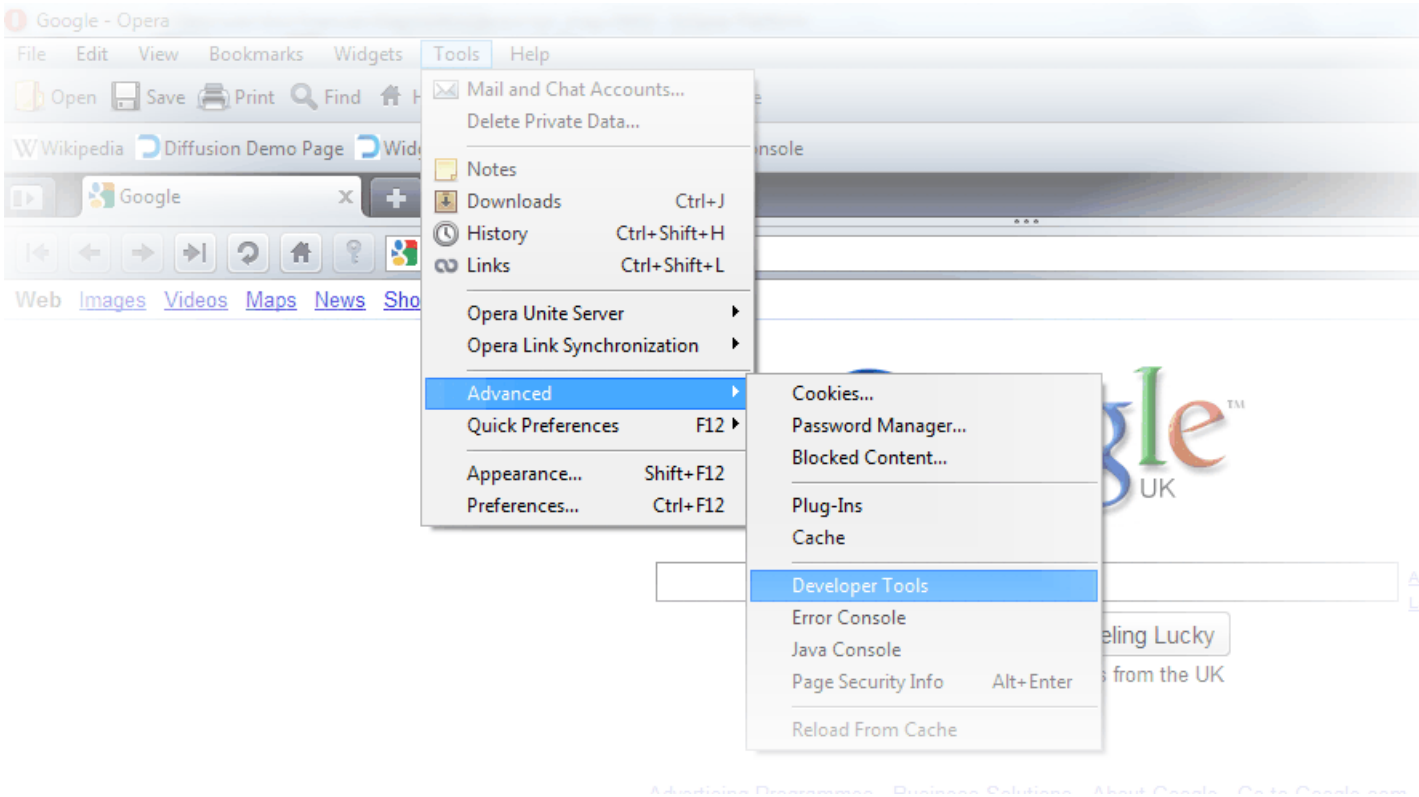
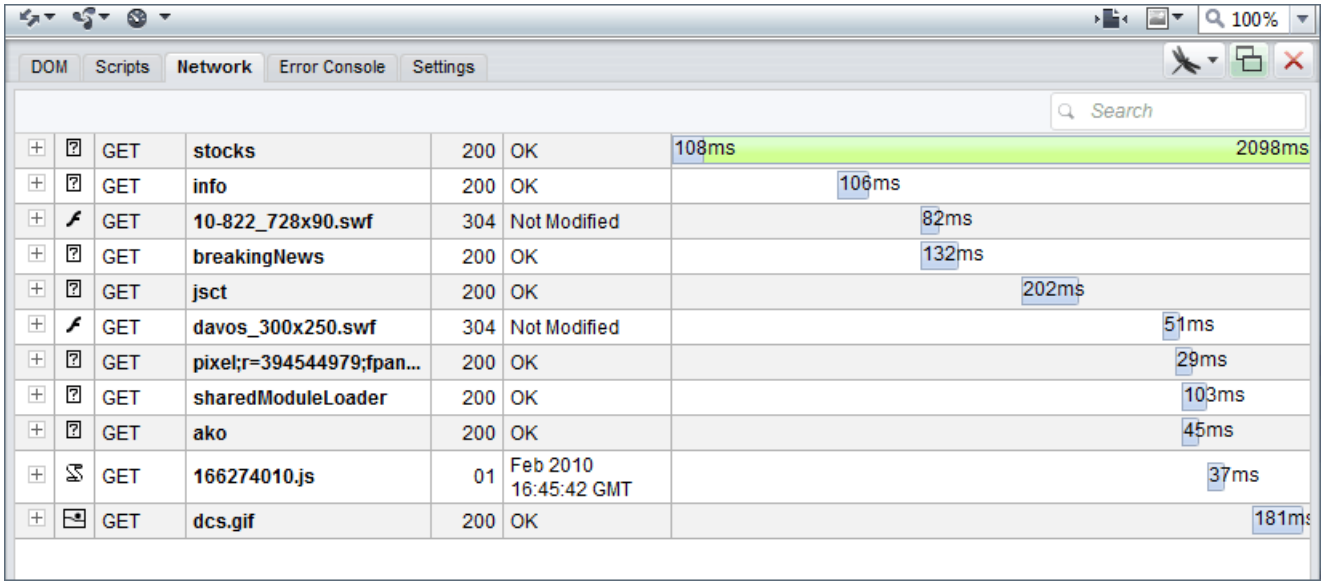


Figure 65: Opera's console

- which will open the console allowing you to begin debugging.


100%
DOM Scripts Network Error Console Settings
Search
GET stocks 200 OK 108ms 2098ms
GET info 200 OK 106ms
GET 10-822_728x90.swf 304 Not Modified 82ms
GET breakingNews 200 OK 132ms
GET jsct 200 OK 202ms
GET davos_300x250.swf 304 Not Modified 51ms
GET pixel;r=394544979;fpan... 200 OK 29ms
GET sharedModuleLoader 200 OK 103ms
GET ako 200 OK 45ms
GET 166274010.js 01 Feb 2010 16:45:42 GMT 37ms
GET dcs.gif 200 OK 181ms

Safari

Copyright 2013 Push Technology

Safari's console is Web Inspector, and functions in a very similar way to Chrome's 'Developer Tools'. Profiling and Resources will need to be enabled at first use

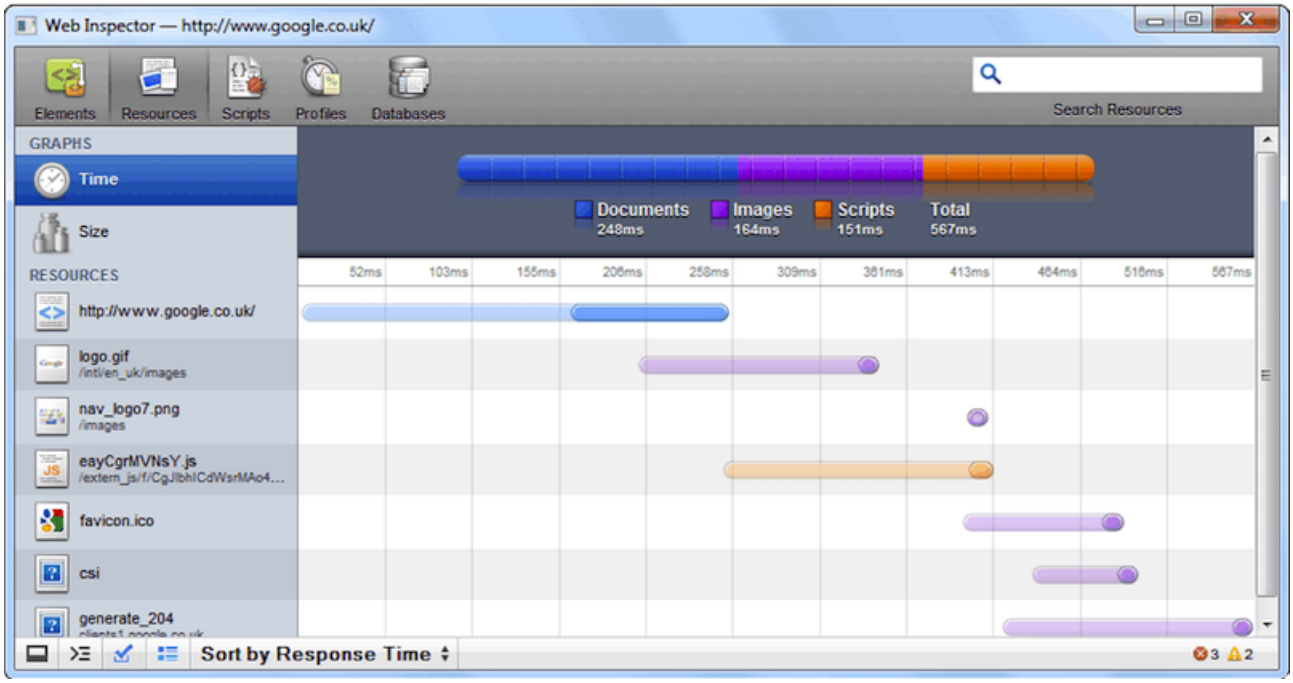


Figure 66: Safari's console

The DiffusionClient library has been compressed. If you want to use the DiffusionClient uncompressed version, use

```
 <script type="text/javascript" href="lib/DIFFUSION/
diffusion.uncompressed.js"></script>
```

Flex and Flash Diagnostics

One of the hardest things about the Flash connectivity is getting the Policy Files correct. If the Policy file isn't loaded or has the incorrect syntax or values it will silently fail. This isn't very helpful, but you can get more information by following the steps below.

Flex / Action Script Diagnostics

Download the debug version for the Flash Player

Flash player debug versions can be found *here*

Create a mm.cfg file

The table below shows where to create a mm.cfg file

Below is an example of entries in a mm.cfg file.

```
# Enables policy file logging
PolicyFileLog=1
# Optional; do not clear log at startup
PolicyFileLogAppend=0
ErrorReportingEnable=1
TraceOutputFileEnable=1
```

Copyright 2013 Push Technology

Viewing the log files

In the following directory there can be two files, flashlog.txt and policyfiles.txt

Windows 95/98/ME	%HOMEDRIVE%\%HOMEPATH%
Windows 2000/XP	C:\Documents and Settings\username
Windows Vista	C:\Users\username
OSX	/Library/Application Support/Macromedia
Linux	/home/username

Windows 95/98/ME/2000/XP	C:\Documents and Settings\username\Application Data \Macromedia\Flash Player\Logs
Windows Vista	C:\Users\username\AppData\Roaming\Macromedia \Flash Player\Logs
OSX	/Users/username/Library/Preferences/Macromedia/Flash Player/Logs
Linux	home/username/Macromedia/Flash_Player/Logs

The link below is a tech note containing more information

http://kb2.adobe.com/cps/193/tn_19323.html

Windows Diagnostics

The .NET APIs and associated client applications produce logging information to log what they are doing and any problems that occur. Logging by default is to a file. The level of logging produced can be adjusted as required with high level logging under normal circumstances or very detailed logging to help with diagnosis of problems.

Configuring API logging

To configure the logging level for the .NET APIs, a developer can tailor the DiffusionLogging class for their own needs. The Logger has a method for each level of logging (severe, warning, etc.).

Log Levels

Supported log levels (in severity order) are as follows:-

- Fatal
- Critical
- Error
- Warning
- Status
- Info
- Debug

Specifying any lower level includes logging for all levels above.

The method you use determines at what level of logging that message would appear. So, by setting the SeverityThreshold to INFO:

```
DiffusionLogging.Logger.SeverityThreshold=Logseverity.Info;
```

This means that only messages for Info and Debug would be logged.

Silverlight Logging Logging Redirection

Copyright 2013 Push Technology

There may be instances where a debug output viewing tool is unavailable; in these circumstances, it is possible to 'redirect' all debug output to the Diffusion™ Debug Console for analysis. This is achieved by setting:

```
Debugging.RedirectDebugOutputToDiffusionConsole = true;
```

in the Debug version of the Silverlight transport.

Debugging a Publisher

Creating a project in Eclipse for debugging publishers.

Having installed Diffusion™ create a Java source directory, and under that directory create a java directory. The directory tree should resemble this;

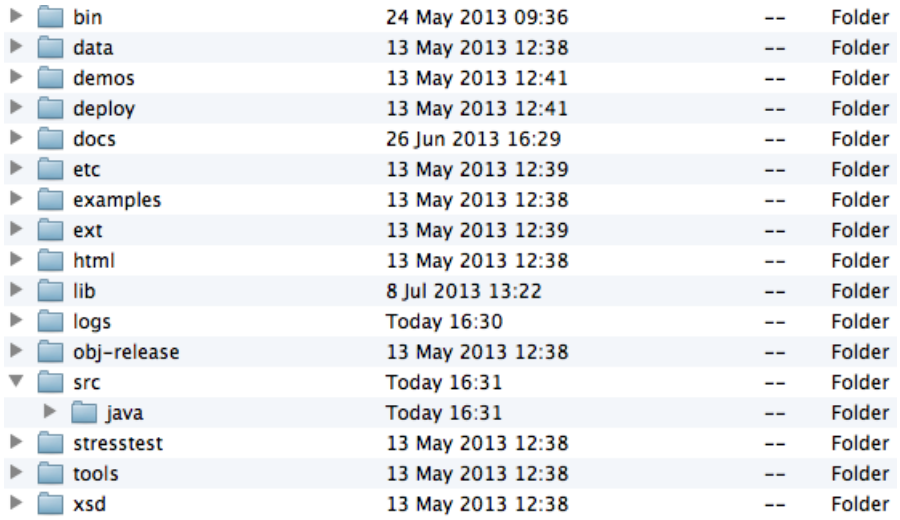

bin 24 May 2013 09:36 -- Folder
data 13 May 2013 12:38 -- Folder
demos 13 May 2013 12:41 -- Folder
deploy 13 May 2013 12:41 -- Folder
docs 26 Jun 2013 16:29 -- Folder
etc 13 May 2013 12:39 -- Folder
examples 13 May 2013 12:38 -- Folder
ext 13 May 2013 12:39 -- Folder
html 13 May 2013 12:38 -- Folder
lib 8 Jul 2013 13:22 -- Folder
logs Today 16:30 -- Folder
obj-release 13 May 2013 12:38 -- Folder
src Today 16:31 -- Folder
java Today 16:31 -- Folder
stresstest 13 May 2013 12:38 -- Folder
tools 13 May 2013 12:38 -- Folder
xsd 13 May 2013 12:38 -- Folder

Figure 67: Example Directory Tree

Create a new Eclipse Java Project and make the project root the src directory.

Copyright 2013 Push Technology

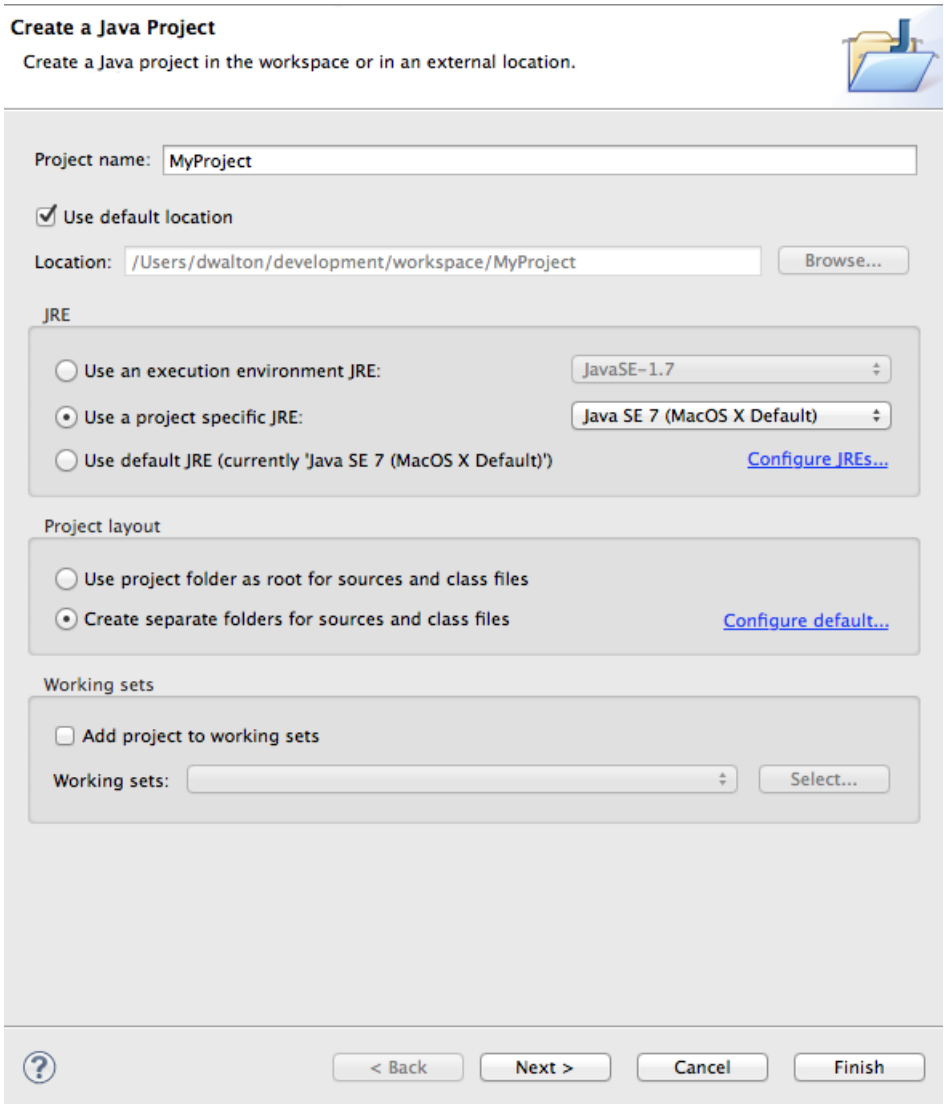
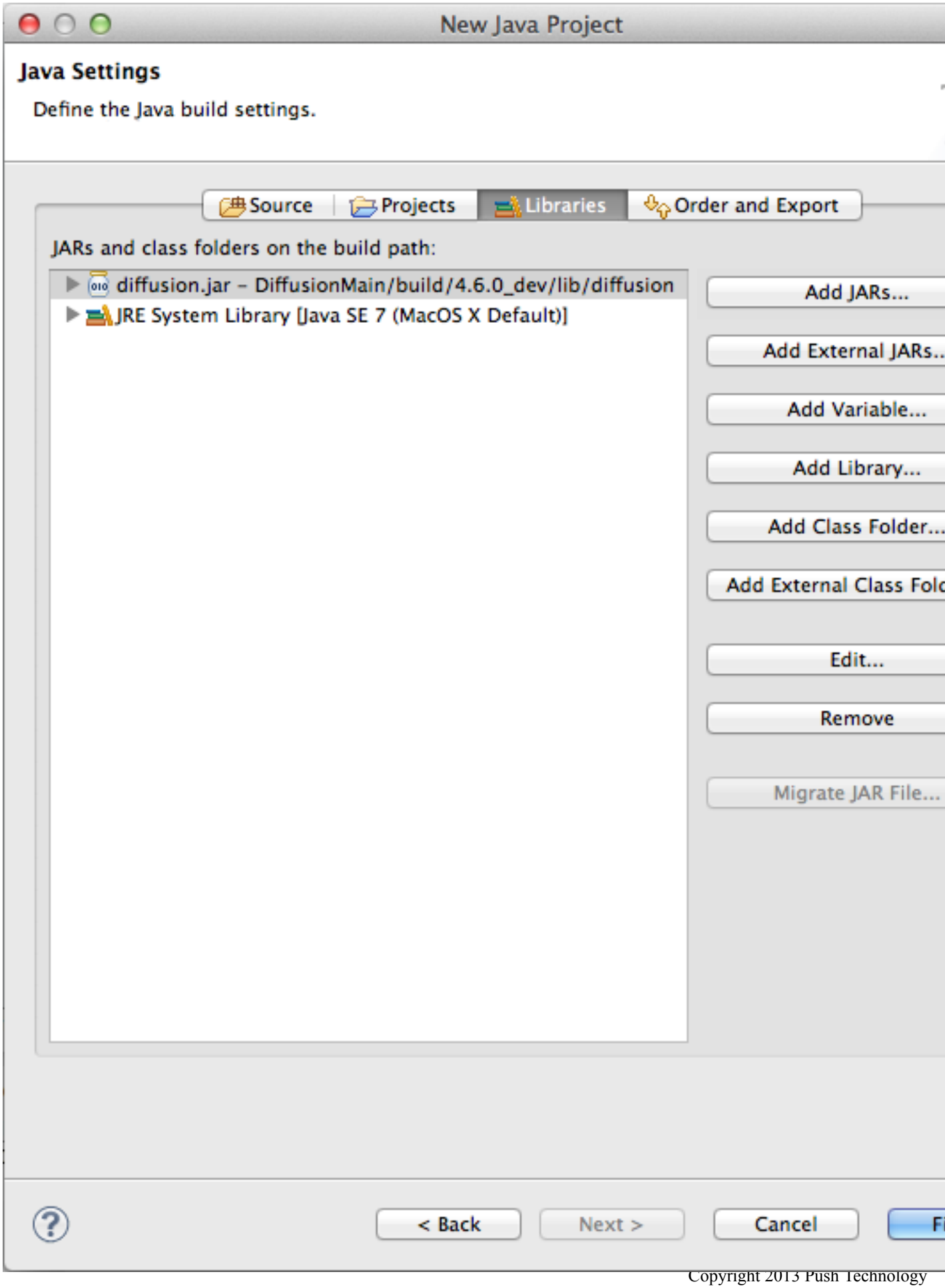


Figure 68: New Java Project

Press **Finish** to create the project. In the **Build Path** add the external jar `lib/diffusion.jar`.

Copyright 2013 Push Technology



Copyright 2013 Push Technology

Figure 69: Build Path Entries

In the java folder create the correct package hierarchy to suit your needs, and then create the your publisher class that extends the class `com.pushtechnology.diffusion.api.publisher.Publisher`

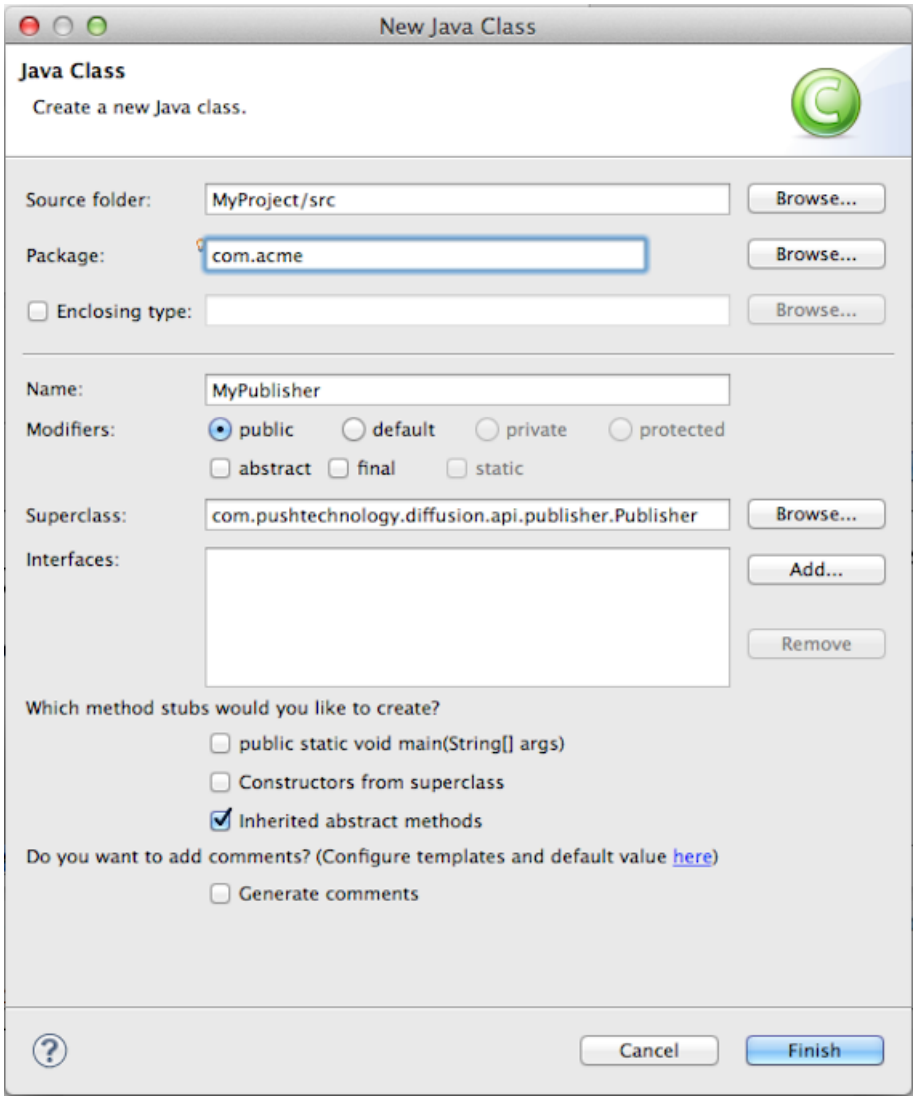


Figure 70: Creating a new Java class

Modify the properties

Modify `etc/Publisher.xml` to load the new publisher. Below is an example of the publisher definition for this publisher.

```
<publisher name="MyPublisher">
 <class>com.pushtechnology.publisher.MyPublisher</class>
</publisher>
```

Create a run target

Open the Run Configurations dialog in Eclipse and create a new run configuration, choose `com.pushtechnology.diffusion.Diffusion` as the main task. In the arguments tab, set the working directory to the `bin` directory of the Diffusion™ Installation. Using the classpath tab, add to the user entries all jar files found your installation's `ext` directory. Finally add the your installations `etc/` folder using the advanced button. You may also need to add any external jar files that are required by your publisher.

Copyright 2013 Push Technology

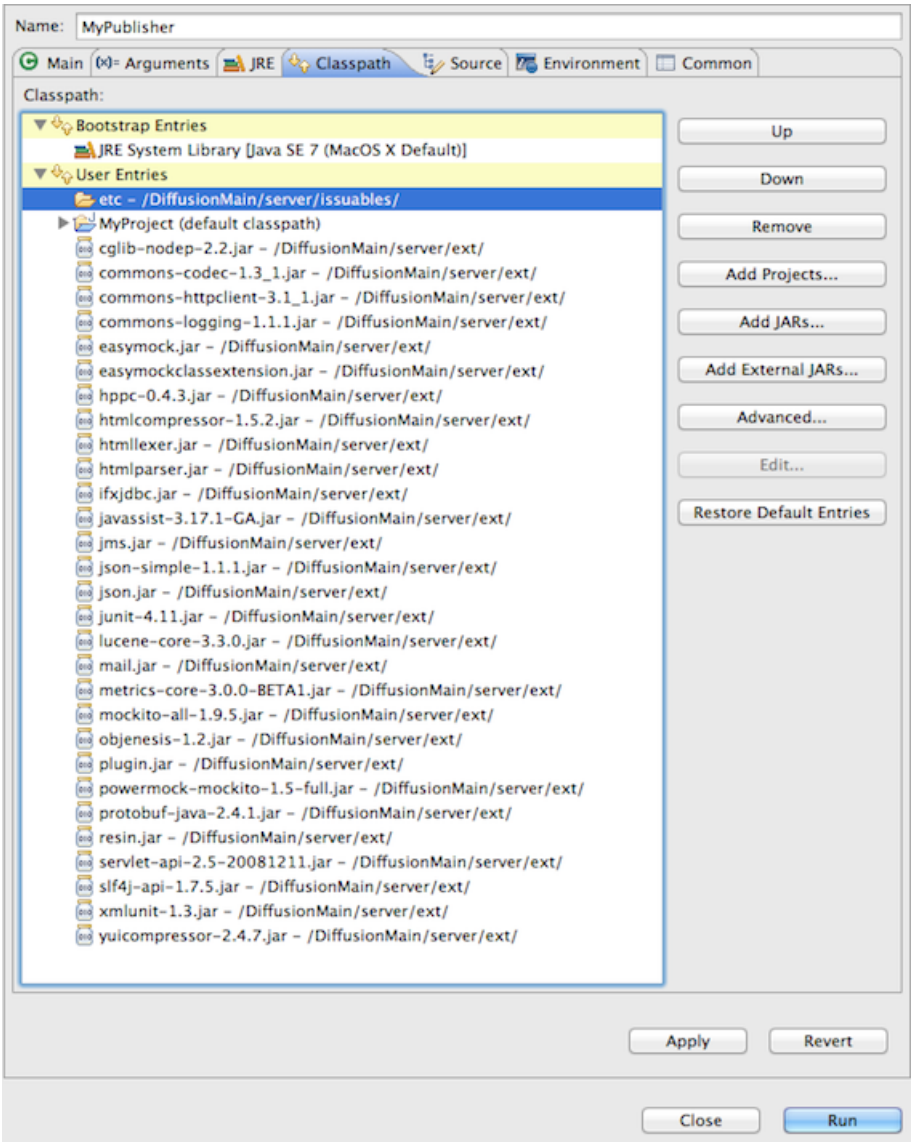


Figure 71: Example classpath entries

NB: Version v4.4 and upward has a new licensing scheme which mandates an extra set of VM argument switches.

```
-javaagent:../lib/licenceagent.jar=../etc/licence.lic,../etc/
publicKeys.store
```

Copyright 2013 Push Technology

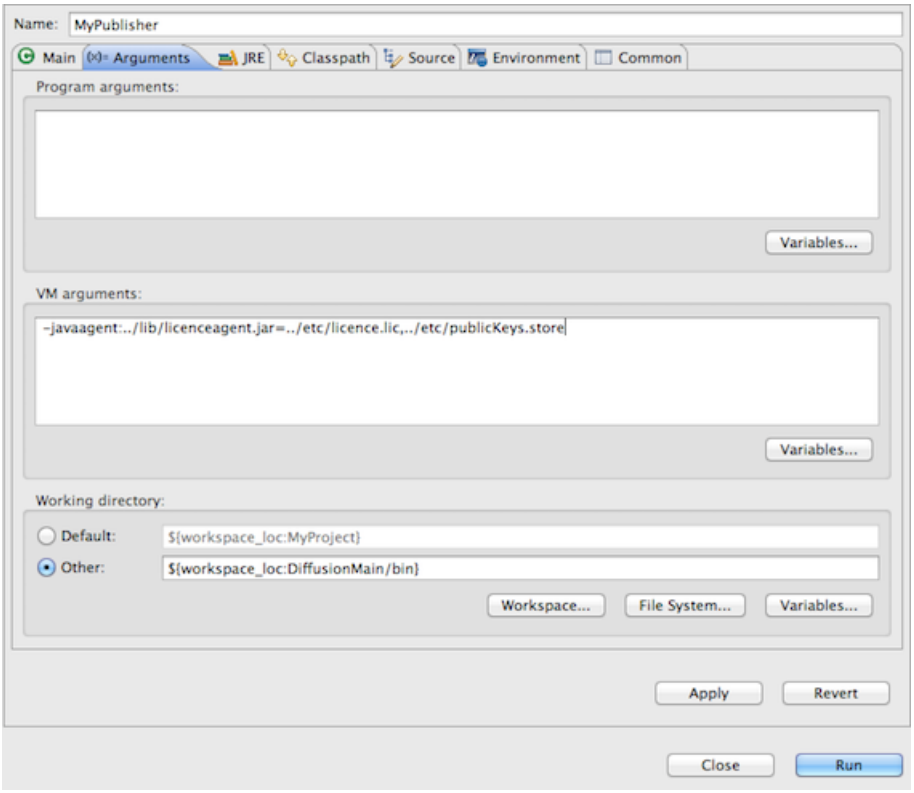


Figure 72: Example Arguments Tab, with licensing switches

You may now set break points within the Publisher. By default Eclipse will create a bin directory in the src directory which will contain the class files for your publisher. By setting the `etc/Server.xml` *usr:lib* entry to this bin directory, when you start Diffusion™ from the command line, your publisher will also be on the class path.

Log Messages

Log Reference

Log Messages

All Server Log Messages

Id	Message	Description
PUSH-000001	Attempt #{} to reconnect to JMS Server.	The JMS adapter is currently disconnected from a JMS server and has just failed another attempt to reconnect.
PUSH-000002	Cannot stop the JMSAdapter: {}.	Something went wrong during initialisation of the JMS adapter, and there was an error when trying to clean up and stop the adapter.
PUSH-000003	Couldn't find JMSReplyTo destination: '{}'.	An attempt has been made to send a message with a JMSReplyTo destination, but the JMS adapter couldn't create the ReplyTo destination on the JMS server.

Copyright 2013 Push Technology

Id	Message	Description
PUSH-000004	Discarding non-TextMessage from JMS: '{}'.	The JMS adapter only supports TextMessage types but it received another message type.
PUSH-000005	Exception from JMS provider '{}'.	The JMS adapter received notification of an exception from the JMS server. Typically, this happens when the connection between the adapter and the server has been terminated.
PUSH-000006	Extra JMS header found, ignoring: '{}'.	JMS messages can have user-defined headers as key/value pairs. Diffusion messages are effectively "value" only, so to map user headers from Diffusion to JMS there must be an even number of them; these are artificially grouped as key/value pairs in the JMS message.
PUSH-000007	Failed to find topic binding for '{}'.	An attempt has been made to send a reply from the JMS adapter back to the JMS server using the DiffusionReplyTo header, but the adapter was unable to determine the corresponding JMSReplyTo destination.
PUSH-000008	Failed to generate delta message '{}'.	A message has been received from the JMS server and a delta message needs to be sent, but the update or publish of the topic data associated with the Diffusion topic failed.
PUSH-000009	Failed to generate ITL message '{}'.	An error occurred while generating an ITL message for a client after a message has been received from a JMS queue.
PUSH-000010	Failed to get message content of message to JMS, topic is '{}'.	Failed to read/parse the content of a Diffusion message while translating it to a JMS message for sending.
PUSH-000011	Failed to send message to JMS destination '{}', exception: {}.	An error occured while sending a message from the JMS adapter to the JMS server.
PUSH-000012	Failed to start JMS Adapter {}.	The JMS adapter was unable to start.
PUSH-000013	Maximum number '{}' of reconnection attempts reached, JMSAdapter disconnected.	The adapter has exhausted the maximum number of configured connection attempts with the JMS server and has given up.
PUSH-000014	Received TextMessage for unknown destination: '{}'.	A message was received by the JMS adapter from the JMS server, but was unable to determine which Diffusion topic it should be mapped to.

Copyright 2013 Push Technology

Id	Message	Description
PUSH-000400	Unable to add listener when reconnecting to {}.	While reconnecting, the JMS adapter was unable to add a listener for a JMS Destination.
PUSH-000401	Unable to create consumer when reconnecting to {}.	While reconnecting, the JMS adapter was unable to recreate a JMS Consumer.
PUSH-000402	Unable to create session when reconnecting to {}.	While reconnecting and re-establishing the existing JMS Consumers, the JMS adapter failed to create a JMS Session.
PUSH-000015	JMS Server disconnect detected, attempting reconnection.	The JMS server has been disconnected from the JMS adapter and the adapter is attempting to reconnect.
PUSH-000016	JMS Server reconnected after '{}' attempts.	The JMS adapter has managed to (re)connect to the JMS server.
PUSH-000017	Unable to create a subscription to '{}', exception is {}.	The JMS adapter failed to create a subscription to the JMS destination associated with the given topic name.
PUSH-000018	Unable to create new topic '{}'.	The JMS adapter has received a message from the JMS server and needs to create a corresponding Diffusion topic, but was unable to.
PUSH-000019	Unable to extract headers from JMS message: '{}'.	Something went wrong while attempting to convert JMS message headers to Diffusion message headers.
PUSH-000020	Unable to get payload from JMS message '{}'.	The JMS adapter has received a message from the JMS server, but is unable to read the content.
PUSH-000021	Unable to restart JMS Server connection.	After an exception from the JMS server, the JMS adapter attempter to reconnect but was unable to.
PUSH-000022	Exception has been thrown by the ServerAckListener.messageNotAcknowledged method of {}.	The specified listener methods has thrown an exception which has been logged but otherwise ignored.
PUSH-000023	Client Auto Failover failed.	A connection to a server has failed, and the autofailover process has encountered an error.
PUSH-000024	Mime extension '{}' was already mapped to '{}' : overwriting map with '{}'.	A configured Mime extension was already mapped to another value but has now been remapped to a new value. This indicates duplicate mime extenstion values in the mimes configuration.

Copyright 2013 Push Technology

Id	Message	Description
PUSH-000025	Connection attempt failed with {} - trying next server.	An attempt to conect to a server has failed but more than one possible server has been specified and so the next is being tried.
PUSH-000026	Requested receive buffer size for HTTP Proxy connection not allocated, requested: {}, allocated {}.	Diffusion tried to set the size of the TCP receive buffer for a socket connection to an HTTP proxy, but the actual number of bytes allocated was different.
PUSH-000027	Requested send buffer size for HTTP Proxy connection not allocated, requested: {}, allocated {}.	Diffusion tried to set the size of the TCP send buffer for a socket connection to an HTTP proxy, but the actual number of bytes allocated was different.
PUSH-000028	An exception has been thrown by the {} method in {}.	An exception has been thrown by an implementation of the specified listener interface. The exception is logged but has no further impact.
PUSH-000029	Remote Service : Client '{}' not known. Inbound message ignored.	A request or notification message has been passed to a Remote Service from the server for a Client that is not known by the service. The message has been ignored.
PUSH-000030	Failure creating a message to pass to the {} listener method.	An error has occurred while converting an inbound request to a message which can be passed to the Remote Service listener.
PUSH-000031	Remote Service Failure parsing Topic Pattern "{}".	A client has attempted to subscribe using an invalid topic pattern.
PUSH-000032	An exception has been thrown by the RemoteServiceListener.{} method.	Remote Service caught an exception thrown by a listener callback method and logged it. The exception has no other effect.
PUSH-000033	Service Request Timeout not supported by server - ignored.	Service Request Timeout specified but is not supported by server version.
PUSH-000034	Remote Service has received subscription request from unknown client {}.	Remote Service has received a subscription request from a client that is not known locally to the Remote Service - request has been ignored.
PUSH-000035	Unable to send fetch reply to Client '{}'.	An error occurred while sending a fetch reply message back to a client.
PUSH-000036	Failure fetching state from '{}' of '{}' : {}.	While handling a fetch request for a topic with topic data, an error occurred while generating a topic load message.

Copyright 2013 Push Technology

Id	Message	Description
PUSH-000037	Subscription Handler '{}' specified for Routing Topic '{}' - ignored.	A subscription handler has been specified for a Routing Topic which is not allowed therefore the handler has been ignored.
PUSH-000038	Exception caught from TopicDeletionListener '{}' topicDeleted method.	An exception has been thrown from a call to a TopicDeletionListener.topicDeleted method. The exception has been logged but has no other effect.
PUSH-000039	Exception caught from TopicTreeListener '{}' {} method.	An exception has been thrown from a call to a TopicTreeListener method. The exception has been logged but has no other effect.
PUSH-000040	WhoIs connection failure limit exceeded - not resolving.	Have attempted to connect to the WhoIs provider 5 times but failed therefore there will be no WhoIs service.
PUSH-000041	Failure to connect to WhoIs provider at {}:{} - will retry 5 times.	Unable to connect to the WhoIs provider indicated - will try 5 more times before giving up.
PUSH-000042	Failed to enumerate files in usr.lib directory '{}'.	An error has occurred while traversing the files within the given directory.
PUSH-000043	Attempted to load duplicate jar file '{}'.	An attempt was made to load the given file more than once.
PUSH-000044	Entry '{}' in server configuration is not a directory.	The given file-system name should identify a directory.
PUSH-000045	Attempt to queue message '{}' for client '{}' when ACK timeout has already expired.	An acknowledgement would be queued, in response to a message requiring an acknowledgement, however the ACK timeout has already expired and is therefore probably too short.
PUSH-000046	Client '{}' closing.	The given client is closing.
PUSH-000047	Client '{}' closing, reason='{}'.	The given client is closing for the specified reason.
PUSH-000048	Error resubscribing Client '{}' to Topic '{}': {}.	An error has occurred while trying to resubscribe the specified Client to the specified Topic.
PUSH-000049	Error subscribing Client '{}' to Topic '{}': {}.	An error has occurred while trying to subscribe the specified Client to the specified Topic.
PUSH-000050	Client '{}' sent command message to Topic '{}' which does not have command-based Topic Data. Message is '{}'.	Command messages can only be processed by topics with related command-based Topic Data.

Copyright 2013 Push Technology

Id	Message	Description
PUSH-000051	Failed to apply Client change event : '{}'.	An error occurred while applying a change to the given client.
PUSH-000052	Exception in ACK listener '{}' messageNotAcknowledged method.	An error occurred while informing a listener that a dispatched message has not been acknowledged.
PUSH-000053	Failed to queue background close of client '{}'.	An attempt to queue the close of a client has failed and therefore the client has been closed in the current thread.
PUSH-000054	Client '{}' tried to fetch invalid topic-selector '{}'.	The topic-selector expression supplied by a Client for a fetch request was invalid.
PUSH-000055	Client Group '"{}"' failed to send message to '{}'.	In sending a message to a Client Group, transmission to the given individual client encountered an error.
PUSH-000056	User licence limit breached, connection rejected.	The maximum number of licensed connections has been exceeded.
PUSH-000057	Server is not licensed to accept distributed connections.	This Diffusion server is not licensed to accept connection from other Diffusion servers..
PUSH-000058	Server is not licensed to accept mobile connections.	The licence does not allow for Diffusion Mobile-to-Server connections.
PUSH-000059	Client '{}' sent message to Topic '{}' to which it is not subscribed.	Diffusion mandates that a client must be subscribed to a topic before it can send messages to it.
PUSH-000060	Unable to subscribe Client '{}' to Topic '{}' - Timed out attempting to acquire lock.	A timeout has occurred while attempting to acquire a lock on a Topic in order for a Client to subscribe to it.
PUSH-000061	Failed to subscribe Client '{}' to Topic(s) '{}' - Invalid Topic name or Selector.	The Client sent an invalid Topic selector pattern for subscription.
PUSH-000062	Failed to schedule future close of Client '{}'.	It was not possible to schedule the future close of a Client that is being kept alive for reconnect purposes.
PUSH-000063	Unable to send fetch reply message to Client '{}' for Topic '{}'.	A failure occurred while trying to send a fetch reply to a Client.
PUSH-000064	Failed to unsubscribe Client '{}' from '{}' - Invalid Topic name or selector.	TThe Client sent an invalid Topic selector pattern for unsubscription.
PUSH-000065	Failure processing message from Server.	An error has occurred at the Client end of a connection to a Server while processing a message from the Server.

Copyright 2013 Push Technology

Id	Message	Description
PUSH-000066	{} closing due to send failure {} {}.	An error occurred while and Event Publisher connection was sending messages.
PUSH-000067	Exception in EventPublisherListener. {} method.	Exception thrown by EventPublisherListener method.
PUSH-000405	Message not sent. The Event Publisher '{}' may not be connected.	The Event Publisher attempted to send with a null Multiplexer. This may be due to the Event Publisher not being connected.
PUSH-000068	Timed out waiting for all messages to be sent.	The EventPublisher connection has been closed while there were messages pending to be sent. The EventPublisher was unable to send those messages in a timely fashion, and they may now be lost.
PUSH-000069	Unexpected error while sending message.	An error occurred when an EventPublisher tried to send a message.
PUSH-000070	Invalid connection from '{}' ('{}').	An invalid connection has been attempted from the given address and location.
PUSH-000071	Unable to rebuild fragmented message.	An error occurred while trying to reassemble message fragments into the original message.
PUSH-000072	Connector '{}' created with support for '{}'.	The given connector was created to support the given types of connection.
PUSH-000073	Connector '{}' does not support Clients or Event Publishers.	A connector has received an attempted connection which indicates that it is either a Diffusion Client or Event Publisher but the connector supports connections from neither.
PUSH-000074	Connector '{}' received an HTTP connection but does not support HTTP.	Connector has received an HTTP connection attempt but does not support HTTP connections (i.e. does not define a valid web server).
PUSH-000075	Unable to start Connector '{}'.	An error occurred while trying to start a connector.
PUSH-000076	Unable to start Connector '{}', permission denied.	The OS denied permission for a resource used by this connector. Often this relates to reserved ports (less than 1024) on Unix based operating systems.
PUSH-000403	'{}': Connection has been closed with an open request	The TCP connection for an HTTP request that had not been responded to has been closed. This may indicate a network problem with the client.

Copyright 2013 Push Technology

Id	Message	Description
		Alternatively it may be caused by the closing of a browser, the close message may be sent to the server after the poll connection has been lost. The client will be closed to indicate that there was an IO problem or that the connection was lost.
PUSH-000077	HTTP Method '{}' is not supported in request from '{}'. Request: '{}'.	An HTTP client has requested a method that is not supported by Diffusion.
PUSH-000404	'{}': No poll has been opened by client within poll timeout	HTTP based connections require an open polling connection to receive messages. If the client has no open polling connection for a fixed period the client will be closed as unresponsive.
PUSH-000078	HTTP Processing error, request from '{}'.	An error occurred in the processing of an HTTP request from the given client.
PUSH-000079	Connector '{}' received a Policy File request which is not supported.	Connector has received a policy file request but does not support such requests (i.e. does not have a policy file configured).
PUSH-000080	Non-SSL connection attempted to connector '{}' which only supports SSL-type connections.	An attempt has been made to make a non secure (SSL) connection to a connector that is configured to only accept secure connections.
PUSH-000081	Connector '{}': no connection - SSL handshake error: {}.	A secure (SSL) connected was attempted but the SSL handshake failed.
PUSH-000082	SSL connection attempted but connector '{}' does not support SSL.	An attempt has been made to make a secure (SSL) connection via a connector that does not support SSL (i.e. does not define a key store).
PUSH-000083	Connector '{}' received a Client connection but does not support Clients.	The given connector is not configured to receive and handle Client connections. The connector may be dedicated to Event Publisher connections, for example.
PUSH-000084	Connector '{}' received an Event Publisher connection but does not support Event Publishers.	The given connector is not configured to receive and handle Event Publisher connections. The connector may be dedicated to Client connections, for example.
PUSH-000085	Requested input buffer size could not be allocated, requested: {} allocated: {}.	The socket receive buffer actually assigned was less than requested (the configured input buffer size). This can have performance implications.

Copyright 2013 Push Technology

Id	Message	Description
PUSH-000086	Requested output buffer size could not be allocated, requested: {} allocated: {}.	The socket send buffer actually assigned was less than requested (the configured output buffer size). This can have performance implications.
PUSH-000087	Connector '{}' received an unidentified connection request [{}] from {}.	An unidentified connection attempt has been made via a connector. The hexadecimal representation of the initial bytes of the connection request are logged in order to aid in identifying the origin.
PUSH-000088	Unable to write rejection response.	An IO error occurred, while sending a rejection response to a client. The client may already have disconnected.
PUSH-000089	Failed to update Child List Topic Data for Topic '{}'.	An unforeseen error occurred while updating this ChildListTopicData.
PUSH-000090	An exception has been thrown from CustomTopicDataHandler.{} method.	The custom TopicData implementation threw an exception in the given method.
PUSH-000091	Client '{}' sent invalid topic selector pattern '{}' to Topic Notify Topic.	A Client has sent an invalid Topic name or selector pattern to a Topic Notify Topic selection.
PUSH-000092	An exception has been thrown from TopicNotifyTopicListener.{} method in '{}'.	The Topic Notify Topic Listener threw an exception in the given method.
PUSH-000093	Topic Notify Select Request Mode '{}' from Client '{}' is invalid.	Client has sent an invalid select request mode to a Topic Notify Topic.
PUSH-000094	{} was unable to notify Client '{}' of Topic '{}' notification event - Client closing.	A failure has occurred notifying a Client of a Topic addition, removal or update - the client will be closed.
PUSH-000095	Unrecognised notification type '{}'.	The notification request holds unexpected content.
PUSH-000096	Client '{}' request to lock Paged Data '{}' timed out.	Unable to action a request from a given client on the specified data as it was not possible to acquire a lock within the timeout period.
PUSH-000097	Paged Topic '{}' Unable to process command '{}' from client '{}'.	A Paged Topic was unable to process a command from a Client.
PUSH-000098	Failure sending a Paged Topic status message to Client '{}'.	A failure occurred while sending a Paged Topic status message to the specified Client.
PUSH-000099	Failure sending a Paged Topic update message to Client '{}'.	A failure occurred while sending a Paged Topic update message to the specified Client.

Copyright 2013 Push Technology

Id	Message	Description
PUSH-000100	Command Topic Data '{}': received invalid command message '{}'	A Client has sent an invalid command message to Command Topic.
PUSH-000101	'{}': received command from client '{}' which is not the edge client.	This relay node received a message from something other than an edge node.
PUSH-000102	Remote Control '{}': could not notify multiple client details to '{}'.	An error occurred while sending multiple client details to a given Remote Service.
PUSH-000103	Remote Control Edge '{}' failed to send connection message to Relay '{}'.	A failure occurred during the connection of a Remote Control Edge to a Relay Server.
PUSH-000104	Remote Control Relay '{}' - error relaying to '{}'.	A Remote Control Relay has encountered an error while relaying a message to a Remote Service.
PUSH-000105	Remote Control Topic '{}': failed to create {} command message.	An error occurred composing a command message for transmission to either the Remote Control relay, or the remote service.
PUSH-000106	'{}': failed to load metadata '{}': {}.	An error occurred parsing the metadata supplied by the remote service.
PUSH-000107	'{}': Failed to notify Client '{}' connection to remote service.	A failure occurred while trying to send a message to a remote service notifying a client connection or resubmission of credentials. When this happens the Remote service will not have been notified.
PUSH-000108	'{}': Failed to notify Client '{}' disconnection to remote service.	A failure occurred while trying to send a message to a remote service notifying a client disconnection. When this happens the Remote service will not have been notified.
PUSH-000109	'{}' failed to report failure to register '{}' by '{}' - failure reason was '{}'.	A Remote Service attempted to register a domain but failed and it was not possible to report the failure back to the Remote Service.
PUSH-000110	'{}': failed to publish data message for topic '{}': {}.	An error occurred publishing a message on the given TopicData-less topic, that was received from a remote service.
PUSH-000111	'{}': failed to publish message for topic '{}': {}.	An error occurred publishing a message on the given topic, that was received from a remote service.
PUSH-000112	'{}': Failed to send fetch reply to client '{}': {}.	An error occurred relaying a fetch-reply message from the remote service to the expecting client.

Copyright 2013 Push Technology

Id	Message	Description
PUSH-000113	'{}': failed to send message to service from client '{}': {}.	An error occurred relaying a command message from the given client to the remote service.
PUSH-000114	'{}': failed to send message to client '{}: {}.	An error occurred relaying a messages from the remote service to the given client.
PUSH-000115	'{}' failed to send service response to client "{}": {}.	An error occurred in Remote Control while relaying a service response to the given client.
PUSH-000116	'{}': failed to subscribe '{}' to Topic '{}': {}.	An error occurred subscribing the given client to the given topic on behalf of a Remote Service.
PUSH-000117	Remote Control failed to send fetch request to '{}' from client '{}' for Topic '{}'.	An instance of Remote Control Topic Data was unable to send a fetch request from a client to the Remote Service.
PUSH-000118	'{}': received message from '{}' for subdomain Topic '{}' which does not have Publishing Topic Data, but has '{}' instead.	A Remote Service Topic has received a message for a subdomain Topic which has the wrong type of Topic Data. Messages can only be sent to Topics with no data or with Topic Data that is of type PublishingTopicData.
PUSH-000119	'{}': received message from client '{}' for topic '{}' which has topic data.	Remote Service subdomain Topic has received a message from a Client on a Topic that has Topic Data. A client can only send messages to subdomain Topics with no Topic Data if they are to be routed to the Remote Service.
PUSH-000120	"{}": received paged topic command from '{}' for topic '{}' which is not a paged topic.	A Remote Service Topic has received a paged topic command for a Topic that is not a Paged Topic.
PUSH-000121	{} failed performing a remove on Paged Topic {}.	A remote Topic was unable to perform a remove on an instance of PagedTopicData.
PUSH-000122	Remote Control '{}' failed to update Paged Topic Data '{}'.	A Remote Control request to updated a Paged Topic has failed.
PUSH-000123	Protocol version mismatch - relay version is '{}' edge version is '{}'.	Remote Control protocol versions must match. Ensure product version consistency to overcome this.
PUSH-000124	Remote Control Topic {} has failed to reconnect to relay after timeout.	A Remote Control Edge Topic timed out waiting for a response from the Relay server and then failed to reconnect to it.

Copyright 2013 Push Technology

Id	Message	Description
PUSH-000125	Remote Control Topic Data '{}' : Relay at '{}' has disconnected.	Remote Control Topic Data has received a notification that a relay server has disconnected.
PUSH-000126	Remote Control Relay Domain '{}' registered.	A Remote Service has registered a Domain with a Relay.
PUSH-000127	Remote Control Relay has lost connection to Edge '{}' : '{}'.	Remote Control Relay has lost connection from the Edge Server.
PUSH-000128	Relay Control Topic at '{}' has been Removed - Relay connection closing.	The relay topic at the relay server has been removed and so the connection to the relay is closing.
PUSH-000129	{} has detected an Exception from the RemoteControlTopicDataListener.{} method of {}.	One of the RemoteControlTopicDataListeners registered with a Remote Control TopicData has thrown an exception.
PUSH-000130	Remote Service Topic '{}' received send to Client '{}' from Remote Service '{}' on Topic '{}' which has Topic Data.	A Remote Service has attempted to send a message to a CLient on a Topic that has Topic Data when this is only permitted on Topics that have no data.
PUSH-000131	Remote Service '{}': closed.	Remote Control Service has been closed.
PUSH-000132	Remote Service '{}' registered with '{}'.	Remote Control Topic Data has accepted the registration of a Remote Service.
PUSH-000133	Remote Service subscribe Clients request for Topic {} which does not exist.	A Remote Service has requested that all Clients be subscribed to a named Topic but that Topic does not exist.
PUSH-000134	Remote Control failed to send subscription request to '{}' from client '{}' for Topic '{}'.	An instance of Remote Control Topic Data was unable to send a subscription request from a client to the Remote Service.
PUSH-000398	Subscription from client '{}' to topic '{}' has failed. The Remote Control has been closed.	An instance of Remote Control Topic Data was unable to send a subscription request from a client to the Remote Service. The Remote Service is closed, you should refer to other logging to find out why.
PUSH-000135	Remote Control Topic was unable to create client notification message.	A Remote Control Topic failed to create a message to notify a Remote Service about a Client. The Remote Service Connection will be closed.
PUSH-000136	{} was unable to obtain lock on {} for update.	A Remote Topic was unable to obtain a lock on an instance of PagedTopicData and therefore could not update it.

Copyright 2013 Push Technology

Id	Message	Description
PUSH-000137	'{}' failed to relay message from '{}' to Edge - service closing.	A failure occurred while trying to relay a message from a Remote Service to the Edge. The Remote Service connection will be closed.
PUSH-000138	Failure passing message from Remote Service to Publisher: {}.	An error has occurred while attempting to pass a message from a Remote Service to the Publisher.
PUSH-000139	Remote Control Relay '{}' failed to send connection acknowledgment to the Edge '{}'.	A Remote Control Relay failed to send a connection acknowledgment to the Edge. Edge connection will not have been established.
PUSH-000140	'{}' failed to send service request from client '{}' to Remote Service.	Remote Control failed to send a service request from a Client to a Remote Service.
PUSH-000141	Remote Service '{}': unregistered.	Remote Control Service has been unregistered.
PUSH-000142	Error removing routing data {} from {}.	A failure occurred while trying to remove a topic from a routing topic.
PUSH-000143	Routing Topic {} failed to route message to client '{}': {}.	A Topic with Routing Topic Data has failed to route a published message to a specific Client.
PUSH-000144	Routing Topic failed to subscribe Client '{}' to Topic '{}'.	A Routing Topic failed to subscribe a Client to a Topic.
PUSH-000145	An exception has been thrown from ServiceTopicListener.{} method in '{}'.	The Service Topic Listener threw an exception in the given method.
PUSH-000146	Service Request error for Client '{}' on Service '{}', request id '{}', error '{}'.	A request from a Client to a Service Topic has failed and it was not possible to report the failure back to the Client, possibly because it is closed.
PUSH-000147	Failure in TopicDataListener '{}' topicDataChanged method	An exception has been thrown by an implementation of the TopicDataListener.topicDataChanged method. The exception is simply logged and has no other effect.
PUSH-000148	Unable to lock Topic {} in order to notify Routing Topic Data instances of Topic removal.	A timeout occurred while waiting to lock the specified Topic in order to notify instances of RoutingTopicData of the Topic removal. It is possible that this error could leave one or more instances of RoutingTopicData in an inconsistent state.
PUSH-000149	{} unable to publish to {}	An update to a Topic has resulted in the publishing to other Topics (e.g. when updating a Topic that has SlaveTopicData instances pointing

Copyright 2013 Push Technology

Id	Message	Description
		at it) and a failure has occurred publishing a message to one of those other Topics.
PUSH-000150	Message '{}' received from '{}' by '{}' has not been handled.	A command Topic that does not handle Messages from Clients has received a message from a Client and ignored it.
PUSH-000397	Deploying publishers in dar file '{}'.	Deploying the publishers in a single DAR file. There may be multiple publishers in a single DAR file. The publishers will be extracted from the DAR file and the additional configuration files loaded.
PUSH-000151	Unable to get JMX MBean attribute.	It was not possible to retrieve an attribute from a JMX MBean.
PUSH-000152	Unable to load shutdown hook class '{}': {}.	An error occurred loading the given 3rd party class.
PUSH-000153	Unable to load startup hook class '{}': {}.	An error occurred loading the given 3rd party class.
PUSH-000154	{} caught JVM shutdown.	The JVM in which Diffusion is executing is being shut down.
PUSH-000155	No connectors have been configured. Creating default connector listening on port {}.	No connectors have been configured therefore a default connector listening on port 8080 has been created with all default values.
PUSH-000156	No Multiplexer Configuration. Creating default configuration.	No Multiplexer configuration has been found therefore a default Multiplexer configuration will be assumed.
PUSH-000157	No queue definitions configured. Creating DefaultQueue.	No queues have been configured therefore a default queue definition has been assumed.
PUSH-000158	No thread pools configured. Created new pool definition called '{}'.	No thread pools were configured therefore a default one has been added.
PUSH-000159	The maximum message size is {} bytes..	The maximum message size has been established.
PUSH-000160	No inbound pool has been configured - using '{}'	No inbound thread pool has been configured therefore the first configured thread pool has been assumed to define the inbound pool.
PUSH-000161	Diffusion - Removing Publishers.	The server is exiting, and all publishers within are being unloaded.
PUSH-000162	Running shutdown hook '{}'.	The given 3rd party class is being executed at server shutdown.

Copyright 2013 Push Technology

Id	Message	Description
PUSH-000163	Running startup hook '{}'.	The given 3rd party class is being executed at server startup.
PUSH-000164	Diffusion Server not started.	Diffusion Server failed to start.
PUSH-000165	Diffusion Server started.	Diffusion started successfully.
PUSH-000166	Diffusion Server '{}' starting.	The Diffusion Server is starting.
PUSH-000167	Diffusion Stopped.	The Diffusion server has been stopped.
PUSH-000168	Diffusion Stopping, reason='{}' by administrator='{}'.	Diffusion is processing a shutdown request.
PUSH-000169	Diffusion - Stopping Connectors.	Connectors are being stopped during a shutdown of Diffusion.
PUSH-000170	Event Publisher {} sent Command message for Topic '{}' which does not have command based Topic Data.	An Event Publisher has sent a command message to a Topic that is not a Command Topic. The message has been ignored.
PUSH-000171	Event Publisher {} sent message for Topic '{}:' which does not exist.	An Event Publisher sent a message for a Topic that does not exist. The message has been ignored.
PUSH-000172	Unable to start StatisticsHandler for Event Publisher '{}'.	An error occurred while registering an Event Publisher with the statistics framework.
PUSH-000173	Exception in '{}' of Listener '{}'.	An exception occurred while processing an internal asynchronous event.
PUSH-000174	Unable to submit event '{}' to thread pool '{}' for execution.	A failure has occurred while submitting a notification event to the specified thread pool for execution.
PUSH-000175	SHA1 algorithm not found, unable to respond to WebSocket request.	The WebSocket protocol requires the SHA1 algorithm, but it was not found. This can only happen with an incomplete or incompatible Java installation.
PUSH-000176	Client '{}' must authenticate. Aborting connection.	An Introspector client has attempted to forgo authentication.
PUSH-000177	Received invalid message '{}' from '{}': {}.	The server module dealing with Introspector clients has received an invalid message.
PUSH-000178	Exception publishing message on topic '{}': {}.	The server module dealing with Introspector clients caught an exception sending a message to the given client.
PUSH-000179	Accepting client '{}': {}.	The server module dealing with Introspector clients has accepted a client connection.

Copyright 2013 Push Technology

Id	Message	Description
PUSH-000180	Topic '{}' may not be removed by Introspector for client {}.	The given client has attempted to remove the given irremovable topic.
PUSH-000181	Topic '{}' not removed for client '{}': {}.	The given topic was not removed for the given client and reason.
PUSH-000182	Topic '{}' removed by Introspector client '{}'.	The given topic was removed for the given client.
PUSH-000183	Message channel error: '{}' caused by '{}'.	A communication error has occurred on a Message Channel.
PUSH-000184	Connector '{}:' has no more Acceptors - Stopping.	An acceptor for a connector has failed, and there are no more acceptors available to service connection requests therefore the connector is stopping.
PUSH-000185	Failed to accept connection for acceptor '{}' on connector '{}'.	This can occur when a socket connection has been made to Diffusion, but a failure occurred while initialising it.
PUSH-000186	Requested receive buffer size not allocated requested: '{}' allocated: '{}'.	Diffusion tried to set the size of the TCP receive buffer for a socket, but the actual number of bytes allocated was different.
PUSH-000187	Second attempt to write output, try increasing timeout or output buffers '{}'.	Diffusion has tried to obtain a selector for writing and failed and is trying again. If this problem persists, try increasing the write timeout or the output buffer size.
PUSH-000188	Selector Thread Failure {} - Closing.	An error has occurred in a selector thread and the thread is closing.
PUSH-000189	Requested send buffer size not allocated, requested: '{}' allocated: '{}'.	An attempt was made to set the sending TCP buffer size for a socket, but the size allocated was not the size requested.
PUSH-000190	SSL unwrap error: '{}'.	An error occurred while decoding SSL-wrapped data.
PUSH-000191	Connector '{}:' - Unable to accept connection.	An exception occurred while attempting to accept a socket connection.
PUSH-000192	Unable to assign an output selector.	Diffusion tried to write a message to a client, but it has temporarily exhausted the number of write selectors that are available. Consider increasing the thread-pools/writer-selectors parameter in Server.xsd.
PUSH-000193	SSL Keystore loaded from '{}'.	SSL Keystore has been loaded from the specified location.
PUSH-000409	Unable to close socket channel '{}'.	The socket channel failed to close.

Copyright 2013 Push Technology

Id	Message	Description
PUSH-000194	JMX cannot expose '{}' as a topic as dimensions exceed 2.	Only 1 and 2 dimensional arrays can be mirrored as a Topic.
PUSH-000195	JMX: Cannot register object '{}' at MBean ObjectName '{}': {}.	An error occurred registering the given object with the given JMX ObjectName with the JMX server.
PUSH-000196	'{}:' failed to update Diffusion licence.	An error occurred while attempting to update the licence file.
PUSH-000197	Unable to create directory '{}' for the public certificates.	The HTTP service which can be used for updating the licence needs to be able to store the licence in a directory, but that directory didn't exist and couldn't be created.
PUSH-000198	Licence is invalid.	The Licence is invalid.
PUSH-000199	Licence is not valid for this version of Diffusion (licence='{}' vs '{}').	The licence file does not match this version of Diffusion.
PUSH-000200	Product '{}' not licensed for this address scheme.".	A product is not licensed for either IP address or MAC addressing schemes.
PUSH-000201	Soft user limit for {} exceeded ({}/{}). Hard limit at {}.	Once the number of connections for a product reaches its soft limit, a warning is emitted. The product may cease to function if this number reaches the hard limit.
PUSH-000202	Licensor: Only '{}' days remaining.	The installed licence file is nearing expiration. A new licence will be required soon in order for Diffusion to continue running.
PUSH-000203	Licence has expired.	The Diffusion licence has expired.
PUSH-000204	Licence hard user limit for {} exceeded ({}/{}).	Once the number of client connections has reached or exceeded a the hard limit, the product may be disabled.
PUSH-000205	Licence for Introspector has expired.	Licence for Introspector has expired.
PUSH-000206	Licensor stopping Diffusion! Invalid licence.	The Diffusion licence is invalid.
PUSH-000207	Licence check failed : Unable to check Server Side Addressing for product [IP] '{}'.	During the licence check, Diffusion attempts to check the server's IP addresses but it was unable to do so.
PUSH-000208	Licence check failed : Unable to check Server Side Addressing for product [MAC] '{}'.	Diffusion is licensed to run on a server with a specific MAC address, but an error occurred while looking for MAC addresses on the server.
PUSH-000209	Licence check failed : Unable to match MAC address for ['{}'].	The product is licensed for a particular MAC address, but the server does not have this MAC.

Copyright 2013 Push Technology

Id	Message	Description
PUSH-000210	Licence check failed : Unable to parse CIDR Address: '{}'.	The product is licensed to run on machines with IP addresses matching a CIDR expression, but the expression could not be parsed. Contact Push Technology for an updated licence file.
PUSH-000211	Licence version is empty, this is not allowed.	Licence version is empty, this is not allowed. Contact Push Technology for a replacement licence file.
PUSH-000212	Log level set to '{} for '{}'.	The logging level of the specified log file has been changed as indicated.
PUSH-000213	Unable to register '{}' with Management Server.	An error occurred while registering the specified object with the JMX Management server.
PUSH-000214	Invalid JMX credentials.	The supplied JMX credentials are incorrect.
PUSH-000215	The management service is disabled.	The Diffusion Management Service is configured to be disabled.
PUSH-000216	The management service has started.	The Diffusion Management Service has started.
PUSH-000217	Unable to Start Management Service.	An error occurred while starting the Diffusion Management Service .
PUSH-000218	Unable to register {} with Management Server - No Management Support.	No management has been configured and therefore it is not possible to register objects. Only the first failure to register an object for this reason will be logged.
PUSH-000219	Failed to cancel fragmented message.	Diffusion took too long to send a fragmented message and tried to send a cancellation message to receiving clients, but a failure occurred.
PUSH-000220	Unable to schedule fragmented message for sending.	A message is being sent in fragments with a delay between each fragment, but Diffusion was unable schedule delivery of the next fragment.
PUSH-000221	Fragment send error.	An error occurred while sending a fragment of a fragmented message.
PUSH-000222	Spent too long trying to send fragmented message with correlation-id '{}', cancelling.	Diffusion has taken too long trying to send fragments of a fragmented message, so the message will be cancelled.
PUSH-000223	Message queue limit {} reached while publishing fragmented message.	A client outbound queue had reached its limit as a result of a message fragmentation operation.
PUSH-000224	Failed to schedule fragmented message for sending.	Unable to schedule the fragmentation of a message before sending.

Copyright 2013 Push Technology

Id	Message	Description
PUSH-000225	Error duplicating multiplexer event message.	An error has occurred duplicating a multiplexer event message.
PUSH-000226	Multoplexer error while processing client '{}'.	A multiplexer tried to send messages for a client, but failed.
PUSH-000227	Multiplexer event is delayed as the queue is full.	An event-driven multiplexer couldn't queue an event due to the queue being full but is continuing to try to do so.
PUSH-000228	Multiplexer event is significantly delayed as the queue is full.	due to the queue being full, an event-driven multiplexer couldn't queue an event after a significant amount of time, but it is continuing to try to do so.
PUSH-000229	Error while handling a multiplexer event.	Error while handling a multiplexer event.
PUSH-000230	Failed to schedule event.	An event-driven multiplexer tried to schedule an event for some time in the future, but wasn't able to do so.
PUSH-000231	Possible latency warning from '{}' [{}ms].	The configured latency warning time has been exceeded while a multiplexer is sending to a client.
PUSH-000232	No multiplexers configured.	No multiplexer definitions have been configured so a default has been assumed.
PUSH-000233	Event-driven multiplexer '{}' started .	An event-driven multiplexer has been started.
PUSH-000234	Properties loaded: '{}'.	The specified properties file has been loaded.
PUSH-000235	Properties reloaded: '{}'.	The specified properties file has been reloaded.
PUSH-000236	Cannot establish JMX attribute '{}' for MBean '{}': {}	The given MBean/Attribute pair is unsupported by Diffusion.
PUSH-000237	Cannot get JMX attribute '{}': {}.	An error occurred while fetching the value of the given MBean attribute.
PUSH-000238	JMX Attribute '{}' for MBean '{}' is unsupported: {}.	The given MBean/Attribute pair is unsupported by Diffusion.
PUSH-000239	Cannot create topic for MBean {} {}.	It was not possible to create a topic to mirror the content of the given JMX MBean.
PUSH-000240	JMX Adapter unable to get topic '{}': {}.	An error in the JMX Adapter while trying to get the given topic from the topic-tree.
PUSH-000241	Cannot remove topic for MBean {}.	It was not possible to remove a topic that mirrors the content of the given JMX MBean .

Copyright 2013 Push Technology

Id	Message	Description
PUSH-000242	JMX Configuring property '{}'={}.	The JMX Adapter configuration is being applied.
PUSH-000243	JMX Cannot parse property '{}': {}.	An error occurred reading the given property.
PUSH-000244	JMX Exception interacting with MBean '{}': {}.	An error occurred creating the tree of topics to represent a JMX MBean.
PUSH-000245	JMX Exception while updating topic '{}': {}.	An error occurred while polling an MBean attribute that the given topic reflects.
PUSH-000246	Caught exception removing notification listener from JMX '{}'.	Diffusion is no longer listening to notification from the given MBean, and this has caused a problem.
PUSH-000247	Cannot find MBean '{}' to listen for notifications.	Diffusion is attempting to add a notificiation to the given MBean, and this has caused a problem.
PUSH-000248	Listener for Mbean '{}' not found.	The notification listener being removed from the given MBean is not currently registered with it.
PUSH-000249	Cannot send message given notification object {}.	A JMX notification was detected and serialised as a message, but it was not possible to send the message.
PUSH-000250	Unexpected notification object '{}'.	A JMX notification of an unexpected type was detected.
PUSH-000251	Unexpected notification type '{}'.	MBeanServerNotification received that was neither about an MBean registering or deregistering.
PUSH-000252	Unable to construct topics for DiffusionPublisher Log Handler: {}.	The Diffusion publisher log handler encountered an error trying to create its topics.
PUSH-000253	Exception caught in {} method of Authorisation Handler {}.	An exception has been thrown by the specified method of the user written Authorisation Handler - the exception has been logged and the requested action rejected.
PUSH-000254	Authorisation manager started with authorisation handler class '{}'.	Authorisation Manager has been started with the specified handler class.
PUSH-000255	Publisher {} connected to Server {} as {}.	A Publisher has successfully connected to another Diffusion server as a client.
PUSH-000256	Deploying publishers compiled for Diffusion version '{}'.	Publishers compiled for Diffusion Version specified are being deployed.
PUSH-000257	Publisher '{}' did not handle message from Server '{}'.	A Publisher has received a message from a server but has not

Copyright 2013 Push Technology

Id	Message	Description
		provided an implementation for the messageFromServer method.
PUSH-000258	Publisher '{}' did not handle server rejected credentials for '{}'.	The default (unimplemented) serverRejectedCredentials method of a Publisher has been called because credentials it has sent to a Server have been rejected.
PUSH-000259	Publisher '{}' did not process message from Event Publisher '{}'.	A Publisher has received a message from an Event Publisher but has not implemented the messageFromEventPublisher method in order to handle it.
PUSH-000260	Diffusion-Version not present in manifest file, not deploying.	Diffusion-Version not present in DAR manifest file so the Publisher(s) will not be deployed.
PUSH-000261	Error deploying Publisher from file '{}'.	An exception has occurred while trying to deploy a Publisher from a DAR file.
PUSH-000262	Error writing file for deployment: {}.	When hot-deploying publishers, the contents of DAR files must be extracted to disk. Check for write permissions to the deployment directory and that there is sufficient free disk space.
PUSH-000263	Exception caught in Publisher '{}' in method '{}'.	A Publisher implementation has thrown an exception in the given method. The exception has been logged but otherwise ignored.
PUSH-000264	Exception caught in publisher '{}' in systemStarted method.	Publishers declared in etc/Publishers.xml can be informed that Diffusion is ready by overriding the systemStarted() method. This message is emitted if an Exception is thrown by the Publisher's systemStarted() method.
PUSH-000265	Failed to load publisher '{}'.	Diffusion was unable to load a publisher, most likely due to a problem with the publisher's configuration files.
PUSH-000266	This dar file is for Diffusion version {} or greater. Not deploying on Diffusion version {}.	The META-INF/MANIFEST.MF of the DAR file contains the "Diffusion-Version" attribute that specifies with which versions of Diffusion it is compatible. The DAR is marked as incompatible with this version of Diffusion and will not be deployed.
PUSH-000267	Introspector Publisher defined, but not licensed.	Diffusion is attempting to start the publisher which is required for the Introspector, but the licence file does

Copyright 2013 Push Technology

Id	Message	Description
		not allow for the Introspector to be run.
PUSH-000268	Publisher '{}' is not stoppable, not undeploying.	An attempt has been to undeploy a publisher, but it's isStoppable() method returns false. The publisher will not be undeployed.
PUSH-000269	JMX adapter threw exception while starting: {}.	An unexpected exception prevented the JMX adapter from starting.
PUSH-000270	Configured not to poll MBeans.	The JMX adapter is configured not to poll those JMX MBean attributes with a matching topic subscription.
PUSH-000271	JMX adapter configured not to start.	The JMX adapter is configured not to start, and therefore will be unavaiable.
PUSH-000272	Polling MBeans every {} milliseconds.	The JMX adapter is configured to configured to poll required MBean attributes for the given frequency.
PUSH-000273	JMX adapter starting.	The JMX adapter has begun construction.
PUSH-000274	DeploymentMonitor: The dar file '{}' has not been fully undeployed. The publisher(s) {} have not been undeployed. The dar file will be ignored from now on.	An error occurred during an attempt to undeploy a publisher, which had been deployed using the "hot deploy" feature. This could happen if a DAR contains multiple publishers, but not all of them return "true" from the isStoppable() method.
PUSH-000275	Publisher '{}' received notification of non-acknowledgement of message '{}' by {} clients.	A publisher has received a notification that one or more Clients have not ackowledged a message requiring acknowledgment, but has not implemented the messageNotAcknowledged method to handle it.
PUSH-000276	Remote Control Publisher {} will connect to relay server at {}:{}.	Indicates the host and port that a Remote Control Publisher is configured to connect to.
PUSH-000277	Loading Remote Control Publisher '{}'.	Remote Control Publisher is loading (initialLoad method).
PUSH-000278	Remote Control Publisher {} has received Message from Remote Service '{}': {}.	A Remote Control Publisher has received a message on the RemoteControlTopicDataListener.messageFromServ method but has not implemented (overridden) that method.
PUSH-000279	Loading Remote Control Relay Publisher '{}'.	Loading Remote Control Relay Publisher.
PUSH-000280	Remote Control Publisher {} registered Remote Service {}.	A Remote Control Publisher has received a message on the

Copyright 2013 Push Technology

Id	Message	Description
		RemoteControlTopicDataListener.serviceRegistered method but has not implemented (overridden) that method so this default message has been logged.
PUSH-000281	Remote Control Publisher {} registered Remote Service {}.	A Remote Control Publisher has received a message on the RemoteControlTopicDataListener.serviceUnregistere method but has not implemented (overridden) that method so this default message has been logged.
PUSH-000282	Removed Publisher '{}'.	The given Publisher has been removed from the server.
PUSH-000389	Server Connection '{}' has been added for Publisher '{}'.	A configured server connection has been successfully added for a Publisher.
PUSH-000390	Failed to add Server Connection '{}' for Publisher '{}'.	SSL keystore file has successfully been loaded for a Publisher to server connection.
PUSH-000283	Publisher to Server connection '{}' closing due to send failure: {}	A connection between a Publisher and a Diffusion Server is being closed due to a failure while sending messages to the Server.
PUSH-000410	Failed to queue background close of the Publisher Server Connection '{}'.	An attempt to queue the close of the Publisher Server Connection failed, therefore the connection has been closed in the current thread.
PUSH-000391	Server Connection '{}' for Publisher '{}' loaded SSL key store '{}'.	A publisher has created a server connection that has loaded a key store to establish the SSL context. The key store is used to securely store key, certificate and trust information.
PUSH-000392	Publisher '{}' has lost connection to Server '{}' - Publisher closing.	A Publisher has lost connection to a server and the recovery policy indicates that the publisher should be closed.
PUSH-000393	Publisher '{}' has lost connection to Server '{}' - no recovery policy specified.	A Publisher has lost connection to a server and no recovery policy has been specified.
PUSH-000394	Publisher '{}' has lost connection to Server '{}' - will retry every {} milliseconds.	A Publisher has lost connection to a server and the recovery policy is to retry therefore a retry will occur at the specified interval.
PUSH-000284	Publisher '{}' unable to connect to Server '{}' - no recovery policy specified.	A Publisher Server connection has failed to connect and no recovery policy has been specified.

Copyright 2013 Push Technology

Id	Message	Description
PUSH-000395	Publisher '{}' unable to schedule retry connection to '{}' - publisher stopping.	A failure has occurred attempting to schedule a connection retry between a Publisher and a server. The publisher will be closed.
PUSH-000285	Started publisher '{}'.	A publisher has been started.
PUSH-000286	Starting publisher '{}'.	A publisher is starting.
PUSH-000287	Failed to start Statistics Handler for Publisher '{}'.	A failure has occurred while trying to start the statistics handler for the apecified Publisher.
PUSH-000288	Stopped Publisher '{}'.	The given Publisher has been stopped, and could be restarted.
PUSH-000289	Stopping Diffusion Server due to stop-server-if-not-loaded flag for publisher '{}'.	Diffusion failed to start a publisher which has the "stop-server-if-not-loaded" configuration value set to "true" in Publishers.xml, so Diffusion will now be stopped.
PUSH-000290	Failure fetching state of {} from Publisher {} {}.	While handling a fetch request for a topic with no topic data, an error occurred while generating a topic load message.
PUSH-000291	Publisher Server Connection '{}' message queue limit reached.	The specified Publisher Server connection is being closed because the number of messages in its outbound queue has reached the configured limit.
PUSH-000411	Multiplexer undefined. The PublisherServerConnection '{}' may not be connected.	The Publisher Server Connection attempted to send with a null Multiplexer. This may be due to the PublisherServerConnection not being connected.
PUSH-000292	Publisher Server Connection {} encountered an unexpected error while sending - closing.	A Publisher Server connection encountered an unexcepected error while sending and will be closed.
PUSH-000293	Publisher '{}' unable to connect to Server '{}' - will retry every {} milliseconds.	A Publisher Server connection could not be established but the recovery policy indicates that it should retry and so a an automatic retry will occur at the interval specified.
PUSH-000294	Failed to read dar file '{}'.	Diffusion tried to deploy a publisher from a DAR file, but the DAR was unreadable. Check file permissions and for corruption with the "jar" tool.
PUSH-000295	Publisher '{}' not currently deployed, cannot undeploy.	An attempt was made to undeploy a publisher which wasn't deployed. Either check that this is the case, or that you have the correct spelling of the publisher's name.

Copyright 2013 Push Technology

Id	Message	Description
PUSH-000296	DeploymentMonitor: One or more of the publishers in '{}' cannot be stopped. Not undeploying file. The dar file will be ignored from now on.	An attempt was made to undeploy a hot-deployed publisher, but that publisher does not return "true" from its isStoppable() method.
PUSH-000297	Output buffer too small ({}). Automatically fragmenting a message to fit; please consider doing this programmatically.	The message being written is larger than the maximum-message-size, and is fragmented to overcome this. This may incur performance penalties.
PUSH-000298	No queue definitions configured : Adding DefaultQueue.	No queue definitions have been configured therefore a default queue configuration is being assumed.
PUSH-000299	A Conflation merge error has occurred in '{}'.	An instance of MessageMerger has thrown an exception from its merge method. The exception is logged and no conflation occurs.
PUSH-000300	Message of size {} is too large for {} which has a limit of {}.	An attempt has been made to send a message to a client that is larger than it is possible to transmit to that client, possibly because of buffer limitations.
PUSH-000301	Default queue definition {} does not exist. '{}' used.	There is no configured queue definition with the name as specified as the default queue therefore the first configured queue is assumed to be the default.
PUSH-000302	Default queue definition not configured. '{}' used.	No default queue definition has been specified in the configuration therefore the first configured queue definition has been assumed to be the default.
PUSH-000303	Queue definition '{}' not known - using default.	The specified queue definition is not known within the configuration therefore the default queue definition has been assumed.
PUSH-000304	Scheduler: failed to schedule: {}.	The scheduler failed to schedule the specified task.
PUSH-000305	Failed to start scheduler.	A failure has occurred while attempting to start the scheduling service.
PUSH-000306	Deployment Service failed to process HTTP request.	The publisher deployment service failed to return an HTTP response to the deploying client.
PUSH-000307	PHP Service failed to send 404 response.	An error occurred in the PHP Service while sending a HTTP 404 response to the web client.
PUSH-000308	Failed to process file.	The PHP service was unable to process a PHP file.

Copyright 2013 Push Technology

Id	Message	Description
PUSH-000309	REST Failed to process service request.	REST Service failed to process a request.
PUSH-000310	Method '{}' from Diffusion Servlet called, not being processed.	The Diffusion Servlet's implementation of the given method does nothing .
PUSH-000311	Diffusion Servlet: {}.	Indicates the startup status of the Diffusion servlet.
PUSH-000312	Client statistics log name '{}' is not configured.	The log named in the Client statistics configuration does not exist in the Logs configuration .
PUSH-000313	Client statistics enabled but no log name supplied.	No log name has been specified in the Client statistics configuration.
PUSH-000314	Unable to start scheduler for client statistics.	An error occurred when scheduling the client connection statistics object for daily execution.
PUSH-000315	Unable to start client connection statistics	An error occurred when scheduling the client connection statistics object for periodic execution, as configured.
PUSH-000316	Failed to update statistics file '{}'.	It was not possible to write the given file.
PUSH-000317	Statistics Manager Failure.	An unexpected error occurred while starting the Statistics Manager.
PUSH-000318	Registering Statistics Reporter '{}'.	A configured statistics reporter of the given name is about to be registered.
PUSH-000319	Failed to register Statistics Reporter '{}'.	An attempt to register the named configured statistics reporter has failed - the reporter has therefore been ignored.
PUSH-000320	Started Statistics Reporter '{}'.	The named Statistics Reporter task has started.
PUSH-000321	Statistics Service failed to load statistics reporter '{}'.	A failure has occurred while attempting to load a configured statistics reporter.
PUSH-000322	Error starting Statistics Service.	An error has occurred while starting the Statistics Service.
PUSH-000323	Background thread pool caught exception executing '{}'	A task being executed by the Background thread pool has thrown an exception.
PUSH-000324	Thread Pool '{}' could not create new thread.	A Thread Pool encountered an error while attempting to create a new thread.
PUSH-000325	Thread Pool '{}' could not create new thread: {}	The specified thread pool failed to create an additional thread.

Copyright 2013 Push Technology

Id	Message	Description
PUSH-000326	Thread Pool {} execution of {} failed.	Failure executing a task in the specified thread pool.
PUSH-000327	Thread Pool {} lower limit notification failed: {}.	The specified thread pool was unable to schedule the notification of the lower threshold limit being reached.
PUSH-000328	Thread Pool {} notification of {} to {} failed.	The execution of a notification to a ThreadPoolNotificationHandler failed.
PUSH-000329	Thread Pool Notification Handler {} loaded for Thread Pool {} with upper threshold of {} and lower threshold of {}.	The specified notification handler has been successfully loaded for the given thread pool.
PUSH-000330	Failed to load thread pool notification handler '{}' for Thread Pool '{}'.	A failure occurred while trying to load a thread pool notification handler.
PUSH-000331	Thread Pool '{}' queue full - aborting execution of '{}'	Specified thread pool queue has become full and the rejection policy is to abort the task. Either the pool maximum size or the queue size should be increased.
PUSH-000332	Thread Pool {} queue full - task running in current thread.	Specified thread pool queue has become full and the rejection policy is to run te task in the current thread, which may be inefficient. Either the pool maximum size or the queue size should be increased.
PUSH-000333	Thread Pool {} was unable to schedule the execution of notification of rejected execution : {}.	The specified thread pool failed to schedule the notification of a rejected task.
PUSH-000334	Thread Pool {} upper threshold notification failed: {}.	The specified thread pool failed to schedule notification of upper threshold reached.
PUSH-000335	Uncaught Exception in thread '{}'.	An uncaught exception has been thrown from a specified thread and logged.
PUSH-000399	Directory '{}' does not exist or is not a directory. Using '{}' as a log directory.	The directory of a log definition or the default log directory does not exist or it is not a directory. In this case the temporary file directory is used instead. This is specified by 'java.io.tmpdir'. If the intended directory does exist the problem may be that the relative path is incorrect because the working directory is not what you expect. The simplest way to resolve this is to use absolute file paths to reference log directories. This may be shown when the log is

Copyright 2013 Push Technology

Id	Message	Description
		created and when checking if it needs to be rotated.
PUSH-000336	Algorithm '{}' not found.	The given cryptographic module could not be found.
PUSH-000337	File Service: [{}] adding virtual hosts {} and mapping to {}.	A virtual host has been added to the webserver's file service.
PUSH-000338	Unable to process alias entry in file '{}': {}.	Diffusion tried to add a new alias to the webserver, but failed.
PUSH-000339	HTMLTopicDataSegment: unable to process topic data for {}.	The webserver tried to include the contents of a Topic's TopicData into the response, but a failure occurred.
PUSH-000340	HTMLIncludeSegment: unable to populate buffer [{}].	The Web Server tried to include the contents of another file in the response, but a failure occurred writing to a buffer.
PUSH-000341	Missing attribute '{}' from DiffusionTag {} [{}].	The given DiffusionTag incorrect omits the given attribute.
PUSH-000342	Topic '{}' has '{}' TopicData instead of Single Value TopicData. DiffusionTag (TopicData) [{}].	The given topic lacks SingleValueTopicData, making it impossible to insert here.
PUSH-000343	Topic {} does not have topic data. DiffusionTag (TopicData) [{}].	The given topic lacks TopicData and therefore any state, making it impossible to insert here.
PUSH-000344	Can't find file attribute in DiffusionTag [{}].	The given DiffusionTag incorrectly omits the 'file' attribute.
PUSH-000345	Cannot load include file '{}' from DiffusionInclude '{}': {}.	An error occurred loading the given file.
PUSH-000346	Cannot read specified file for DiffusionTag '{}' [{}].	An error occurred loading the given file specified in the DiffusionTag.
PUSH-000347	Unknown publisher in DiffusionTag {} [{}].	The Publisher named in this DiffusionTag is not loaded, or does not exist.
PUSH-000348	Unknown topic for DiffusionTag (TopicData) {} [{}].	The Topic named in this DiffusionTag does not exist.
PUSH-000349	Unknown type attribute '{}' for DiffusionTag [{}].	The type attribute of this DiffusionTag holds an unexpected value.
PUSH-000350	Web Server service cannot process HTML tag '{}' for the publisher '{}'.	An error occurred while processing an HTML embedded tag.
PUSH-000351	{}: Connection rejected as request was missing the HTTP header field '{}'.	This connection was rejected as it was missing the given HTTP header. The connection is likely not from a Diffusion client, or an intermediary (firewall/load-balancer) has removed it.

Copyright 2013 Push Technology

Id	Message	Description
PUSH-000352	{}: connection rejected as [{}] did not match '{}'.	The connection from an untrusted host was rejected, in accordance with CORS configuration.
PUSH-000353	{}: Connection rejected as [{}] did not match {}.	The connection was rejected because it's websocket origin did not match the regular expression stored in the configuration.
PUSH-000354	{}: cors-origin not defined or invalid. Aborting request.	CORS (Cross Origin Resource Sharing) request has been received by the Web Server Client Service but no "cors-origin" has been configured or the configured value was invalid. The request has been aborted.
PUSH-000355	{} unable to process HTTP Request '{}'.	Web Server File Service was Unable to process the given HTTP Request.
PUSH-000356	{}: HTTP Processing error on connector '{}': {}.	An error occurred handling an HTTP request.
PUSH-000357	{}: Invalid regular expression '{}' for CORS origin '{}' - feature disabled.	An invalid regular expression was given for the CORS origin configuration.
PUSH-000358	{}: Invalid regular expression '{}' for WebSocket origin '{}' - feature disabled.	An invalid regular expression was given for the WebSocket origin configuration.
PUSH-000359	Poll request for client '{}' from '{}' failed as no matching client can be found. Client may have been disconnected previously. Poll request will be ignored.	The Diffusion Web Server has received a poll request for a Client that is not known to the server. This may be because the Client has been closed by the Server. This may also indicate an incorrectly configured load balancer, routing an HTTP connection to a server that was not servicing the given client. Consider enabling session-stickiness for HTTP Diffusion clients.
PUSH-000360	FileService [{}]: not adding virtual hosts '{}' mapping to '{}' as it is not a directory.	Failed to add a virtual host to the webserver's file service because the configured mapping does not point to a directory.
PUSH-000361	Unable to service HTTP request for service '{}'.	An HTTP service was invoked, but it failed to process the incoming request.
PUSH-000362	Started HTTP Service for '{}'.	An HTTP service was successfully started.
PUSH-000363	{} Unable to create log.	An HTTP service has its own log file defined, but Diffusion was unable to create the logger.

Copyright 2013 Push Technology

Id	Message	Description
PUSH-000364	Web Server {} failed to instantiate HTTP service '{}' : {}.	A web server failed to instantiate the named configured HTTP service.
PUSH-000365	Web Server failed to start cache service for virtual host '{}'.	An error occurred while starting the cache service for the given virtual host.
PUSH-000366	Virtual Host HTTP request {} is invalid.	An HTTP request being handled by a virtual host is trying to break ot of the virtaul root.
PUSH-000367	Virtual Host HTTP request {} - unable to read file {}.	A failure has occurred trying to read the specified HTML file.
PUSH-000368	{}: websocket-origin not defined or invalid. Aborting request.	Web Server Client Service has received a Web Socket request but "websocket-origin" has not been configured or is invalid. The request has been aborted.
PUSH-000369	Failed to schedule a periodic tidy of the WhoIs cache.	A failre has occurred while trying to schedule a periodic tidy of the WhoIs cache.
PUSH-000370	WhoIs Provider class '{}' instantiation failure.	Unable to instantiate the configured WhoIs provider. Will use the default WhoIs provider.
PUSH-000371	Failed to load WhoIs provider class '{}'.	Unable to load the configured WhoIs provider class. Default WhoIs provider class will be used.
PUSH-000372	Unable to load GeoIP Database '{}'.	An error occurred while loading or initialising the GeoIP database.
PUSH-000373	WhoIs Service failed to lookup address '{}'.	A failure has occurred in the WhoIs service when trying to look up the given address.
PUSH-000374	WhoIs service starting with {} thread(s).	The WhoIs service is starting.
PUSH-000375	Deprecated connector type RTMP found in Connectors.xml and will be ignored.	Deprecated connector type RTMP found in Connectors.xml and will be ignored.
PUSH-000376	Deprecated element '{}' found in {}.xml.	A deprecated element has been found in specified properties file and should be removed.
PUSH-000396	Unable to perform substitution within Property Value '{}' due to syntax error.	A syntax error has been detected whilst trying to perform environment variable substitution on a configuration property.
PUSH-000377	Failed to load XML properties '{}'.	Failed to load XML properties.
PUSH-000378	Failed to load publisher '{}' configuration from XML properties.	Diffusion was unable to load the configuration for the named

Copyright 2013 Push Technology

Id	Message	Description
		publisher from an XML properties file.
PUSH-000379	Failed to load subscription validation policy '{}' from file '{}' : {}.	A failure has occurred loading an XML subscription validation policy file.
PUSH-000380	XML Validation Event {} : {}.	An XML validation event has occurred while validating an XML property file.
PUSH-000381	Loaded XML {} properties from '{}'.	The specified XML properties have been loaded from the file indicated.
PUSH-000382	Unable to load Publisher config file {}.	Attempted to load a publisher's configuration file (Publishers.xml), but it couldn't be found or is unreadable.
PUSH-000383	unable to check server-side addressing for product {} [{}] : {}.	An error occurred while attempting to check server side addressing in the licence file.
PUSH-000384	'{}': Failed to decode licence key '{}'.	Unable to decode licence key.
PUSH-000385	Failed to load subscription validation topic mapping for Publisher {} topic {} from file {} : {}.	A failure has occurred loading a publisher topic mapping for a subscription validation policy.
PUSH-000386	Failed to load connection validation policy {} from {}.	A failure has occurred loading XML Connection Validation Policy from the specified file.
PUSH-000387	Licence check failed to match MAC address for '{}'.	Unable to match a MAC address for the specified product.
PUSH-000388	Failed to save XML Properties '{}'.	A failure has occurred while saving XML properties.

Copyright 2013 Push Technology

Chapter 22

Introspector

Topics:

- *Supported Platforms.*
- *Installing from update site*
- *Installing subsequent plugin updates.*
- *Deinstallation*
- *Finding the Diffusion™ perspective*
- *Adding servers*
- *Opening servers*
- *Exploring the topics*
- *Getting topic values*
- *Configuring columns*
- *Ping servers*
- *Count topics*
- *Using the Clients view*
- *Ping*
- *Statistics*
- *Topics*
- *Logging*
- *Server logs*
- *Property obfuscator*

An introduction to the Introspector Eclipse plugin

This section introduces and explains the features of the Diffusion™ Introspector GUI. This is currently a beta-product, and as such may contain bugs.

The Introspector GUI is a set of Eclipse plugins, and therefore any user should already be au-fait with using Eclipse. This product depends on a minimum of Eclipse Juno (or v4.2) which can be downloaded from http://www.eclipse.org/downloads/

This GUI uses features from the existing System Publisher as well as the new Diffusion™ Publisher for which a licence must be obtained.

Supported Platforms.

The stack of technologies upon which this software is expected to function.

Introspector is tested on Eclipse Juno (v4.2), and JDK1.7.

From February 2013 onwards Oracle are no longer providing public updates to JDK1.6.

Installing from update site

All Diffusion™ Eclipse plugins are distributed primarily via the Eclipse Update-site *http://download.pushtechnology.com/eclipse/4.6* To install the software, ensure you have Eclipse Juno installed at least. Lesser version may work, though your mileage may vary. Pick the 'Help' menu, click the 'Install new software' menu item. From the dialog, choose the 'Add ...' button. Complete the dialog as shown below.

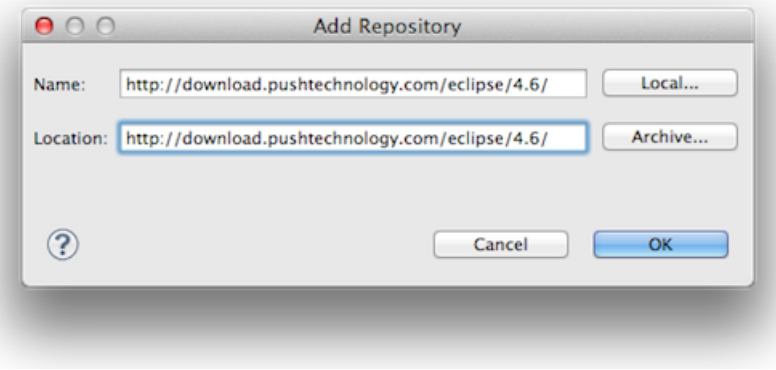


Figure 73: Adding a Repository

Click on the 'Ok' button to see the dialog shown below

Copyright 2013 Push Technology

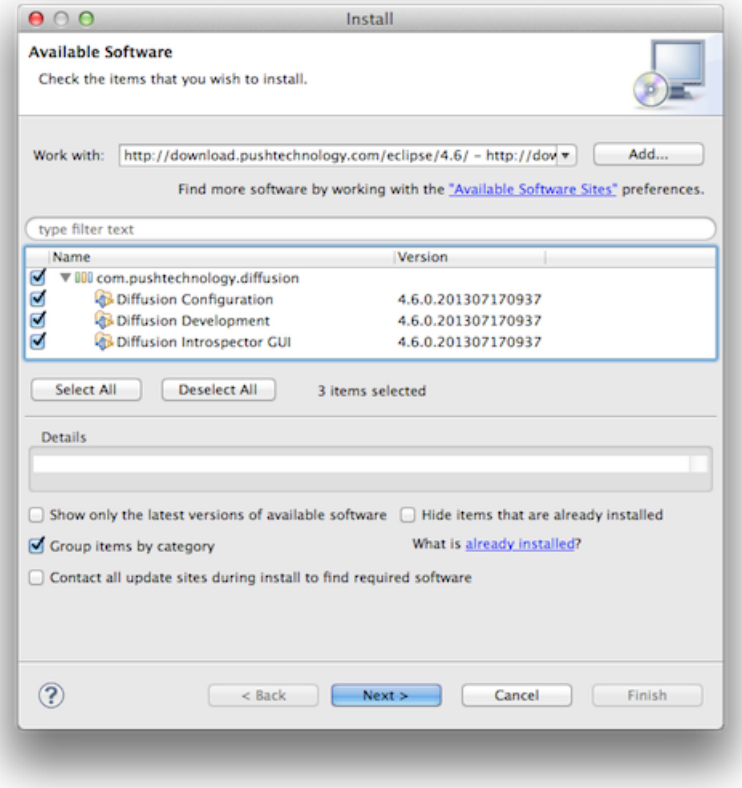


Figure 74: Next step...

Click the 'Next' button. Follow the sequential stream of dialogs. When you are given the dialog shown below, be sure to click on 'I accept the terms of the license agreement' so that you can proceed.

Copyright 2013 Push Technology

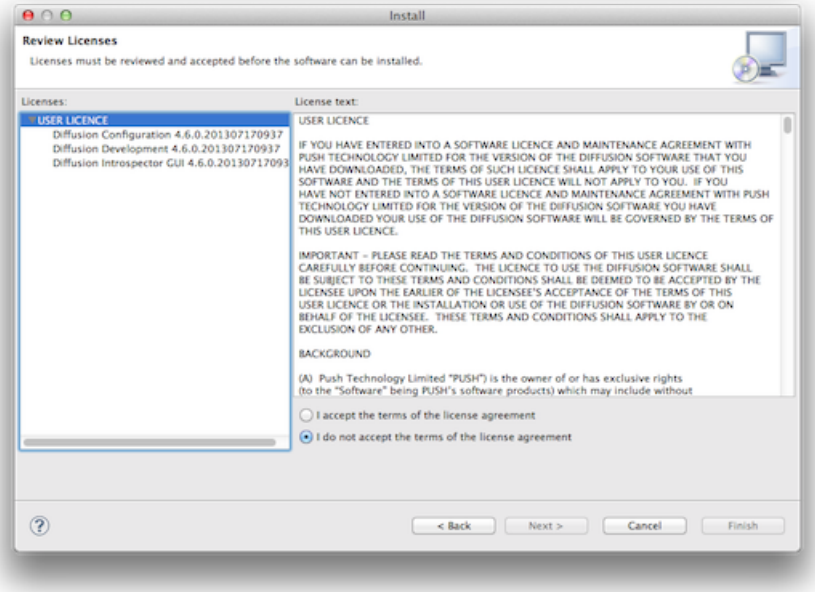

Install
Review Licenses
Licenses must be reviewed and accepted before the software can be installed.
Licenses:
License text:
USER LICENCE
Diffusion Configuration 4.6.0.201307170937
Diffusion Development 4.6.0.201307170937
Diffusion Introspector GUI 4.6.0.201307170937
USER LICENCE
IF YOU HAVE ENTERED INTO A SOFTWARE LICENCE AND MAINTENANCE AGREEMENT WITH PUSH TECHNOLOGY LIMITED FOR THE VERSION OF THE DIFFUSION SOFTWARE THAT YOU HAVE DOWNLOADED, THE TERMS OF SUCH LICENCE SHALL APPLY TO YOUR USE OF THIS SOFTWARE AND THE TERMS OF THIS USER LICENCE WILL NOT APPLY TO YOU. IF YOU HAVE NOT ENTERED INTO A SOFTWARE LICENCE AND MAINTENANCE AGREEMENT WITH PUSH TECHNOLOGY LIMITED FOR THE VERSION OF THE DIFFUSION SOFTWARE YOU HAVE DOWNLOADED YOUR USE OF THE DIFFUSION SOFTWARE WILL BE GOVERNED BY THE TERMS OF THIS USER LICENCE.
IMPORTANT - PLEASE READ THE TERMS AND CONDITIONS OF THIS USER LICENCE CAREFULLY BEFORE CONTINUING. THE LICENCE TO USE THE DIFFUSION SOFTWARE SHALL BE SUBJECT TO THESE TERMS AND CONDITIONS SHALL BE DEEMED TO BE ACCEPTED BY THE LICENSEE UPON THE EARLIER OF THE LICENSEE'S ACCEPTANCE OF THE TERMS OF THIS USER LICENCE OR THE INSTALLATION OR USE OF THE DIFFUSION SOFTWARE BY OR ON BEHALF OF THE LICENSEE. THESE TERMS AND CONDITIONS SHALL APPLY TO THE EXCLUSION OF ANY OTHER.
BACKGROUND
(A) Push Technology Limited "PUSH") is the owner of or has exclusive rights (to the "Software" being PUSH's software products) which may include without
I accept the terms of the license agreement
I do not accept the terms of the license agreement
< Back
Next >
Cancel
Finish

Figure 75: Accept the license agreement

When you are shown the dialog below, be sure to click on the 'Ok' button, which is not the default, hence the emphasis.

Security Warning
Warning: You are installing software that contains unsigned content. The authenticity or validity of this software cannot be established. Do you want to continue with the installation?
OK
Details >>
Cancel

Figure 76: Click OK...

When the installation process is done, click the 'Restart Now' button.

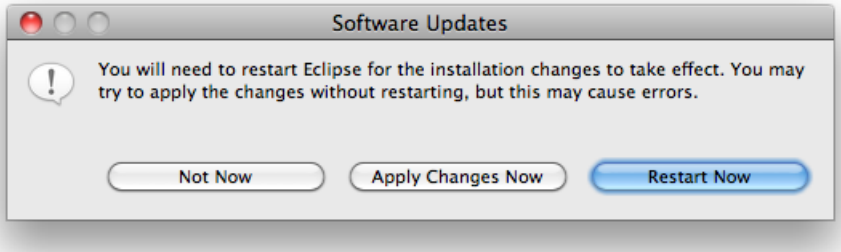

Software Updates
You will need to restart Eclipse for the installation changes to take effect. You may try to apply the changes without restarting, but this may cause errors.
Not Now
Apply Changes Now
Restart Now

Figure 77: Restarting

Copyright 2013 Push Technology

Installing subsequent plugin updates.

Once this process is complete users can install the latest version of the plugins by clicking on the 'Help' menu and picking the 'Check for updates' menu item.

Deinstallation

Eclipse maintains a registry of what plugins have been installed, and gives users a means of removing plugins from it.

Firstly, open the 'About Eclipse' dialog. On the Mac this can be found in the 'Eclipse' menu, on every other platform - the 'Help' menu. It should look something like this

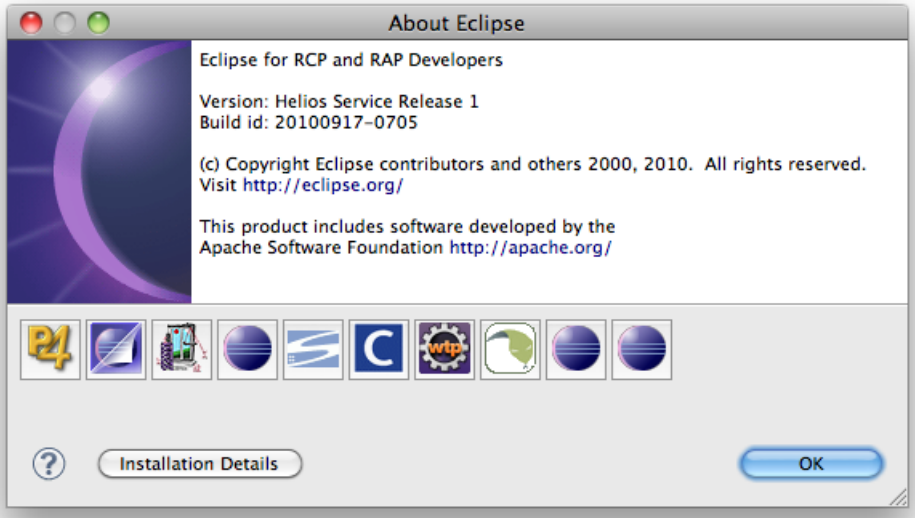

About Eclipse
Eclipse for RCP and RAP Developers
Version: Helios Service Release 1
Build id: 20100917-0705
(c) Copyright Eclipse contributors and others 2000, 2010. All rights reserved.
Visit http://eclipse.org/
This product includes software developed by the
Apache Software Foundation http://apache.org/
Installation Details
OK

Figure 78: "About Eclipse" dialogue

Click on Installation details. Once the 'Eclipse Installation Details' dialog appears, click on the 'Installed Software' tab to review which plugins are installed.

Copyright 2013 Push Technology

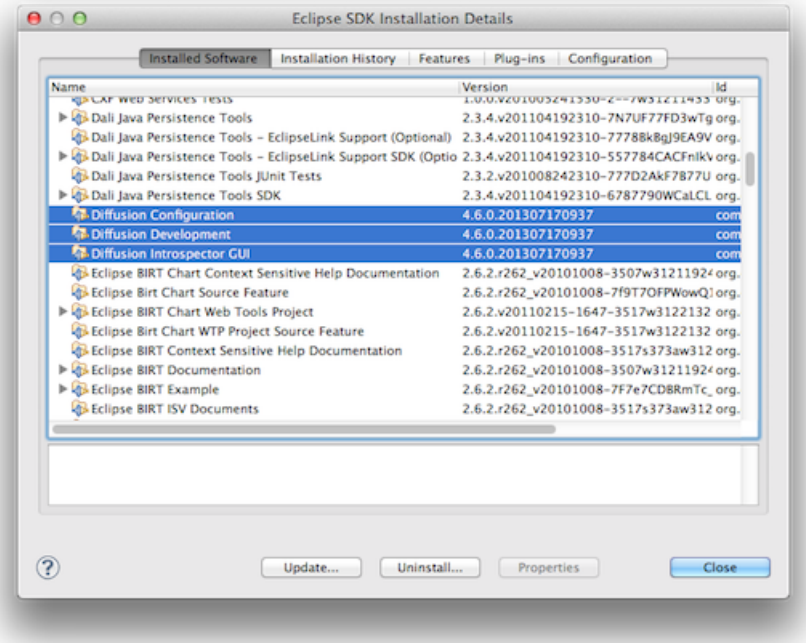


Figure 79: Installed Plugins

Once a the software is found, you may uninstalled it using the 'Uninstall ...' button. Users may be advised to restart Eclipse once the process is complete.

Finding the Diffusion™ perspective

From the Window menu pick "Open Perspective" and "Other...". Pick the 'Diffusion' item from the list and click on 'Ok'.

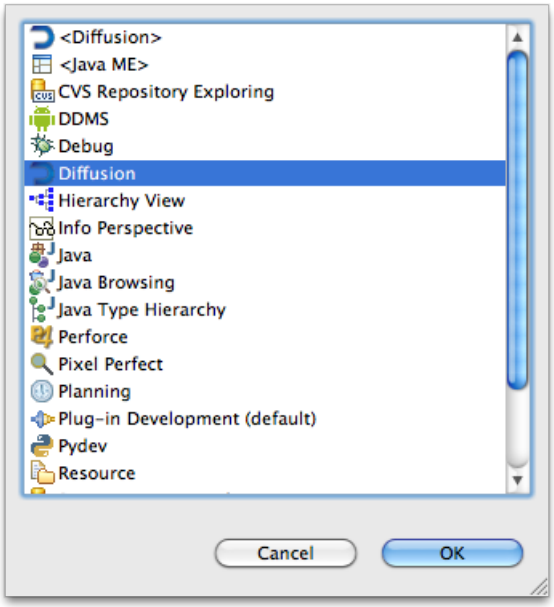


Figure 80: Perspective

Copyright 2013 Push Technology

Opening the Diffusion™ perspective will present to the user a set of views related to Diffusion™ functionality. The same views can be added to other perspectives (such as the Java perspective):

- From the Window menu pick "Show View" and "Other".
- Find the Diffusion™ section in the list and pick the views required.

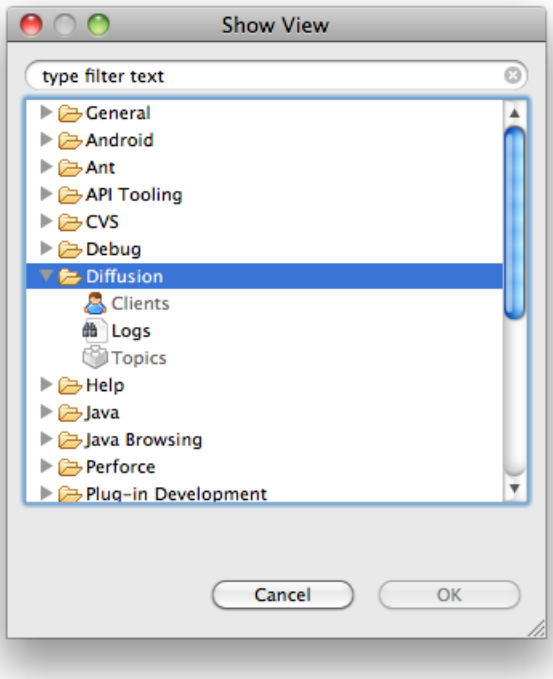


Figure 81: Views

The Diffusion™ perspective shows the Topics and Clients views on the left hand side.

Adding servers

To add server details to the plugin click the

button in the head of the Topic view or Client view. The user may click on the 'Test' button if they wish to test the worthiness of the connectivity details before clicking Ok.

Copyright 2013 Push Technology

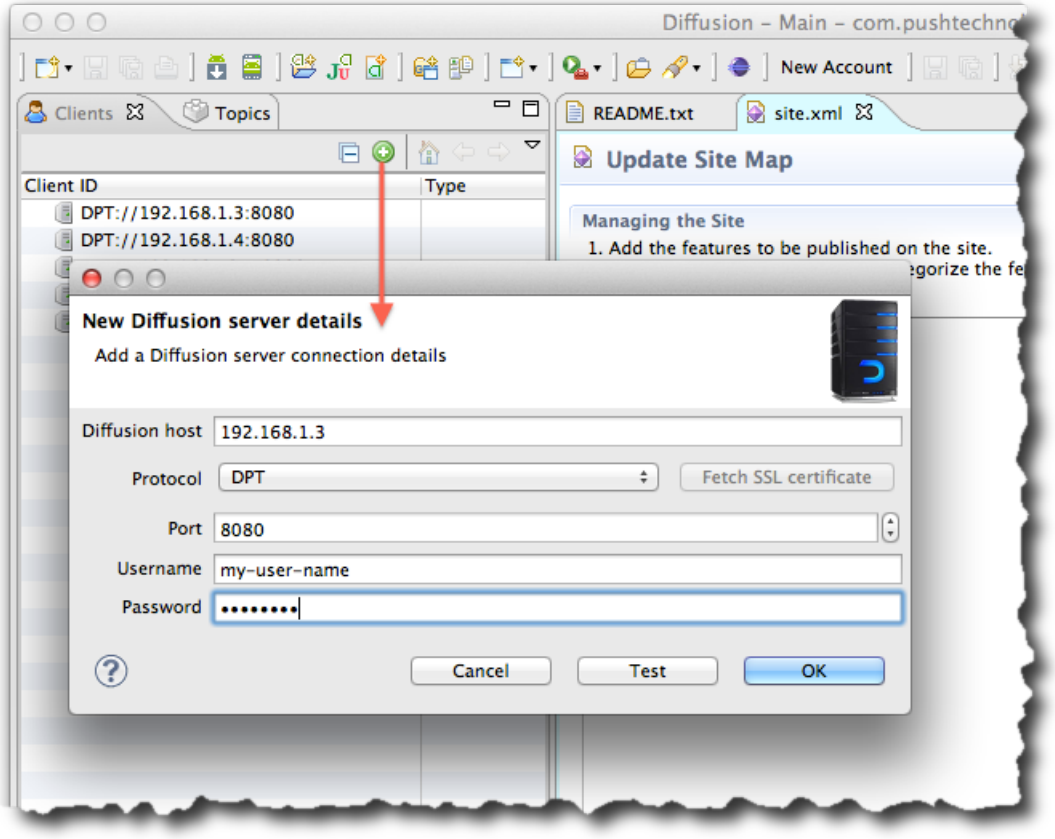


Figure 82: Add a Server

If the user needs to amend server details, the context menu of closed server (i.e. those that are not currently connected) holds an 'Edit' item that presents the same dialog box.

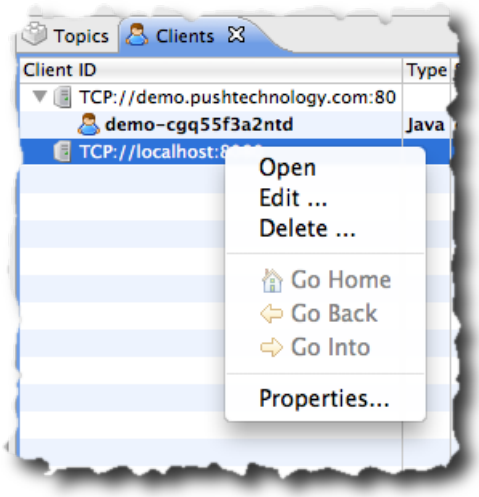


Figure 83: Edit Server Details

Opening servers

Copyright 2013 Push Technology

In either the Topics view or the Clients view, right-click on a server icon and pick 'Open' from the context menu, or double click.

The set of columns shown by default show the Diffusion™ server name (server.name in *etc/Server.xml*) in the value column and the time that the server was started in the 'Created' column.

Exploring the topics

Users can navigate the tree of topics as they would any other tree control. The GUI populates the tree on-demand and asynchronously, so it's possible that a branch of the tree may populate some short time after it has been opened. Each topic shows it's name, the publisher, the type of any topic-data associated, the date-stamp the topic was created, the date-stamp the topic was last updated and the value of that topic. The last two will only be populated if the topic is fetched or subscribed (see below).

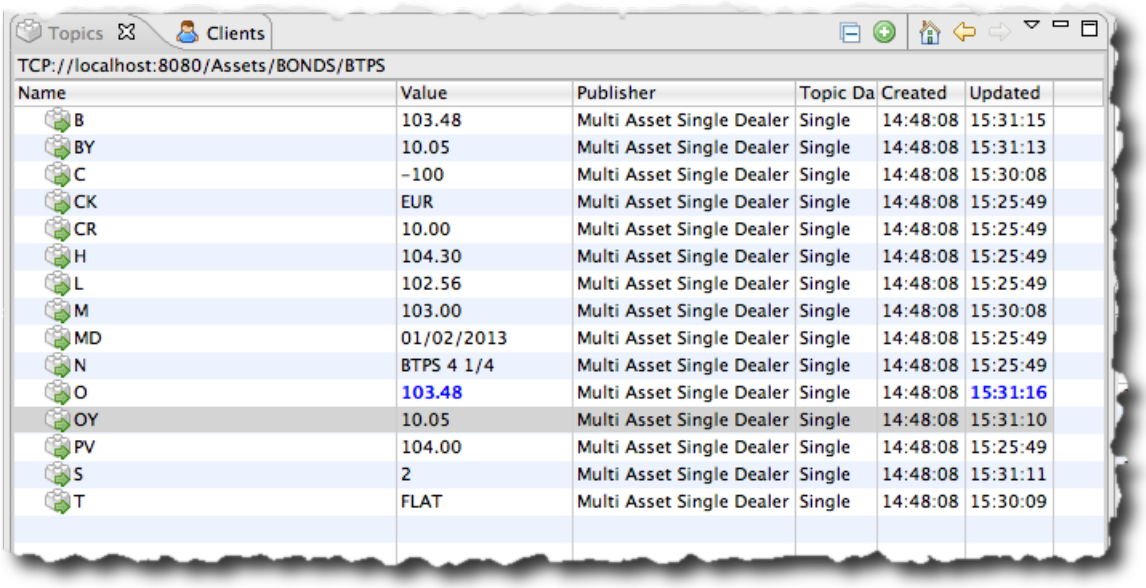
The topic tree is live and as new topics are created and old topics deleted so the tree will be updated.

Getting topic values

Users may retrieve the current state of the topic by picking 'Fetch' from the context menu of a topic. Picking 'Fetch recursively' will also fetch the values of all topics below this one.

Users may also subscribe to a topic by picking 'Subscribe' from the context-menu of a topic. 'Subscribe recursively' will also subscribe to values of all topics below this one. When the value of a topic changes the Value and Updated columns flash blue for a second.

If the user wishes to drill into a part of the topic tree they may pick 'Go Into' from the topic context menu. The view then shows the name of the root topic-name in the view header.



Name	Value	Publisher	Topic Da	Created	Updated
B	103.48	Multi Asset Single Dealer	Single	14:48:08	15:31:15
BY	10.05	Multi Asset Single Dealer	Single	14:48:08	15:31:13
C	-100	Multi Asset Single Dealer	Single	14:48:08	15:30:08
CK	EUR	Multi Asset Single Dealer	Single	14:48:08	15:25:49
CR	10.00	Multi Asset Single Dealer	Single	14:48:08	15:25:49
H	104.30	Multi Asset Single Dealer	Single	14:48:08	15:25:49
L	102.56	Multi Asset Single Dealer	Single	14:48:08	15:25:49
M	103.00	Multi Asset Single Dealer	Single	14:48:08	15:30:08
MD	01/02/2013	Multi Asset Single Dealer	Single	14:48:08	15:25:49
N	BTPS 4 1/4	Multi Asset Single Dealer	Single	14:48:08	15:25:49
O	103.48	Multi Asset Single Dealer	Single	14:48:08	15:31:16
OY	10.05	Multi Asset Single Dealer	Single	14:48:08	15:31:10
PV	104.00	Multi Asset Single Dealer	Single	14:48:08	15:25:49
S	2	Multi Asset Single Dealer	Single	14:48:08	15:31:11
T	FLAT	Multi Asset Single Dealer	Single	14:48:08	15:30:09

Figure 84: View Topic Values

Click the back button to return to the previous position in the topic tree.

Configuring columns

Copyright 2013 Push Technology

Columns can be reordered with drag and drop. Column positions are stored between uses. The context-menu of the headers lets users toggle individual columns on and off.

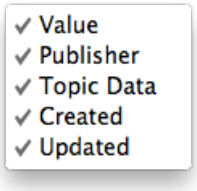


Figure 85: Re-order Columns

Ping servers

To establish the response time of a remote server pick 'Ping' from the context menu of a connected server. The results in milliseconds is shown next to the server name in the Name column.

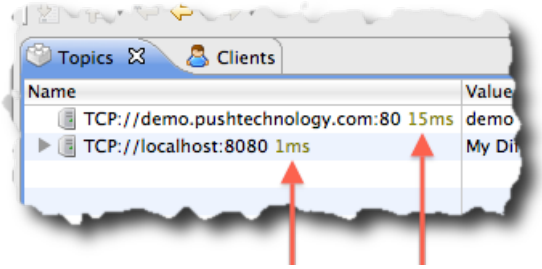


Figure 86: Ping a Server

Count topics

Pick 'Count topics' from the context-menu of a topic for the server to count the number of topics underneath this one. The result is shown to the right of the topic name

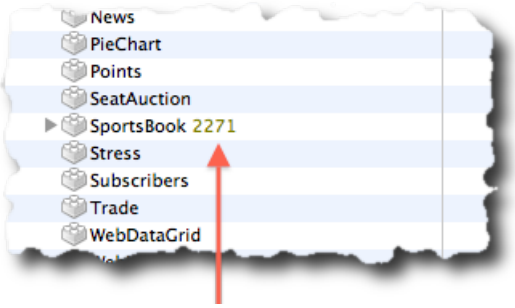


Figure 87: Topic Count

Using the Clients view

The Clients view is a live view of the clients connected to the set of Diffusion™ servers. As the Diffusion™ GUI is also a Diffusion™ client, it will show up in this list and be shown in bold. A newly connected client is bright blue for 1s, while a disconnecting client is drawn pale grey for 1s before it vanishes.

The properties of a client show are

Copyright 2013 Push Technology

Type	Flash, Silverlight, Websocket, iPhone or Android, etc
Server	the name of the server this client is connected to
Start time	the time-stamp the client connected to its server
Resolved name	the result of a WHOIS query on the IP address of the remote client
Description	more details from the WHOIS query
Locale	The nation or territory from which this client is connecting
Ping	The time taken to ping the remote client
Alarm	Not currently in use
IP Address	The IP address of the remote client
Client reference	The reference/description granted to a client

Ping

All Diffusion™ clients respond to ping requests from the server. Using this feature the user may ping individual clients, the results of which show up in the 'Ping' column.

Further details of a client can be gained from the properties view. Pick 'Properties' from the context menu of a client.

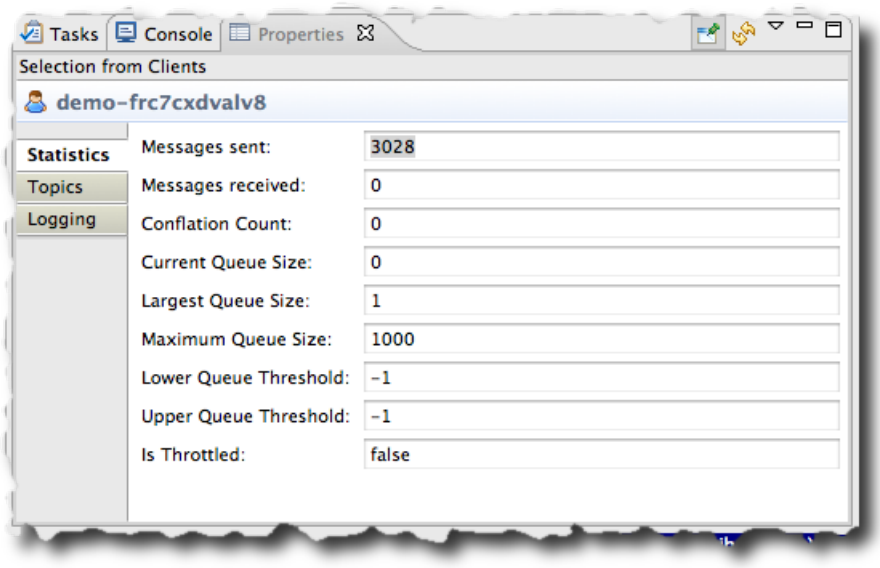


Figure 88: Ping clients

The Properties view is part of the Eclipse framework and used in many other non-Diffusion™ circumstances. Consequently, if the user clicks a new object in another view the Properties view may show the properties of that instead. The properties of a remote client break down into three sections

Statistics

Copyright 2013 Push Technology

This shows the number of messages sent, received, the number of messages conflated, the current queue size, the highest queue size, the maximum queue size allowable and throttling thresholds. Unlike previous views this data is not live, and the refresh button in the view header must be clicked to refresh this view.

Topics

This section shows the topics to which this client is currently subscribed along with any related topic reference. This data is also not live, and must be refreshed manually.

Logging

Access to remote client logs is currently an experimental and optional feature. Those clients that do support it subscribe to the Diffusion/ClientLogs topic. Using this section of the view, it is possible to retrieve log entries from remote JavaScript clients that are normally sent to the JavaScript console alone.

The Javascript environment inside a modern browser is memory constrained. This facet of the GUI lets the user set the level at which a log entry is stored and the size of the buffer in which entries are stored before being removed.

Server logs

Right click on any open Diffusion™ server item in the Topics or Clients view and pick 'Logs ...' from the context menu.

Diffusion™ server logs are published via the Diffusion/Logs topic as well as written to log files. The Logs view subscribes to Diffusion™ to display the log entries as they happen.

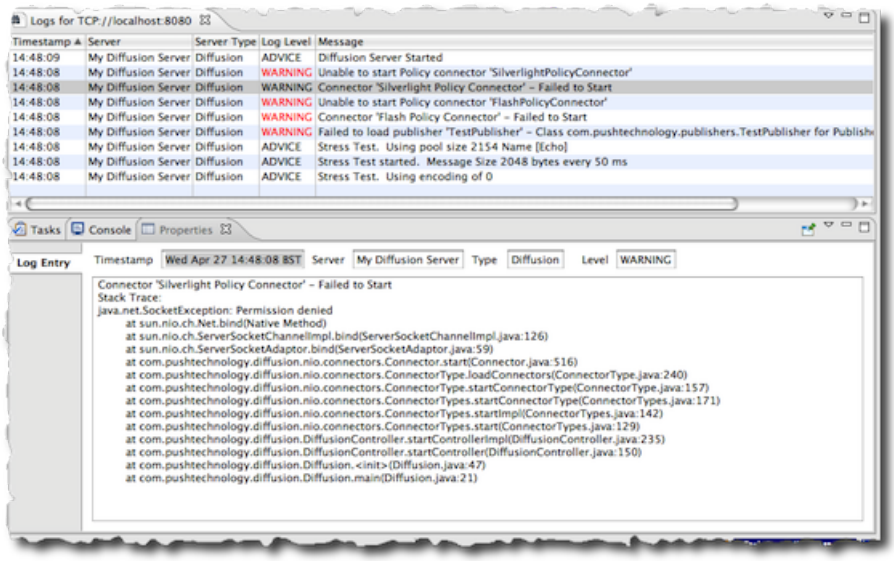

Logs for TCP://localhost:8080
Timestamp
Server
Server Type
Log Level
Message
14:48:09 My Diffusion Server Diffusion ADVICE Diffusion Server Started
14:48:08 My Diffusion Server Diffusion WARNING Unable to start Policy connector 'SilverlightPolicyConnector'
14:48:08 My Diffusion Server Diffusion WARNING Connector 'Silverlight Policy Connector' - Failed to Start
14:48:08 My Diffusion Server Diffusion WARNING Unable to start Policy connector 'FlashPolicyConnector'
14:48:08 My Diffusion Server Diffusion WARNING Connector 'Flash Policy Connector' - Failed to Start
14:48:08 My Diffusion Server Diffusion WARNING Failed to load publisher 'TestPublisher' - Class com.pushtechnology.publishers.TestPublisher for Publish
14:48:08 My Diffusion Server Diffusion ADVICE Stress Test. Using pool size 2154 Name [Echo]
14:48:08 My Diffusion Server Diffusion ADVICE Stress Test started. Message Size 2048 bytes every 50 ms
14:48:08 My Diffusion Server Diffusion ADVICE Stress Test. Using encoding of 0
Tasks
Console
Properties
Log Entry
Timestamp
Wed Apr 27 14:48:08 BST
Server
My Diffusion Server
Type
Diffusion
Level
WARNING
Connector 'Silverlight Policy Connector' - Failed to Start
Stack Trace:
java.net.SocketException: Permission denied
at sun.nio.ch.Net.bind(Native Method)
at sun.nio.ch.ServerSocketChannelImpl.bind(ServerSocketChannelImpl.java:126)
at sun.nio.ch.ServerSocketAdaptor.bind(ServerSocketAdaptor.java:59)
at com.pushtechnology.diffusion.nio.connectors.Connector.start(Connector.java:516)
at com.pushtechnology.diffusion.nio.connectors.ConnectorType.loadConnectors(ConnectorType.java:240)
at com.pushtechnology.diffusion.nio.connectors.ConnectorType.startConnectorType(ConnectorType.java:157)
at com.pushtechnology.diffusion.nio.connectors.ConnectorTypes.startConnectorType(ConnectorTypes.java:171)
at com.pushtechnology.diffusion.nio.connectors.ConnectorTypes.startImpl(ConnectorTypes.java:142)
at com.pushtechnology.diffusion.nio.connectors.ConnectorTypes.start(ConnectorTypes.java:129)
at com.pushtechnology.diffusion.DiffusionController.startControllerImpl(DiffusionController.java:235)
at com.pushtechnology.diffusion.DiffusionController.startController(DiffusionController.java:150)
at com.pushtechnology.diffusion.Diffusion.<init>(Diffusion.java:47)
at com.pushtechnology.diffusion.Diffusion.main(Diffusion.java:21)

Figure 89: Server log entries

By clicking on the "Timstamp" table header Users may order the log entries by time-stamp. The pull down menu on the right hand side of the view header lets the user filter log entries by the entry log-level. NB: this does not affect the logging from the server as this is a client-side filter only. It is possible to get the properties of server log entry from the context menu of an entry. Shown above are the properties of a multi-line log-entry.

Copyright 2013 Push Technology

Property obfuscator

This dialog is part of the Diffusion perspective and can be used to hide sensitive Diffusion™ configuration file entries, such as passwords and JMS login credentials. Select or enter into the dialog the text that you want to obfuscate. All obfuscated entries start with `OB:` . The Copy button will will place the obfuscated text on the pasteboard, ready for pasting back into the configuration file.

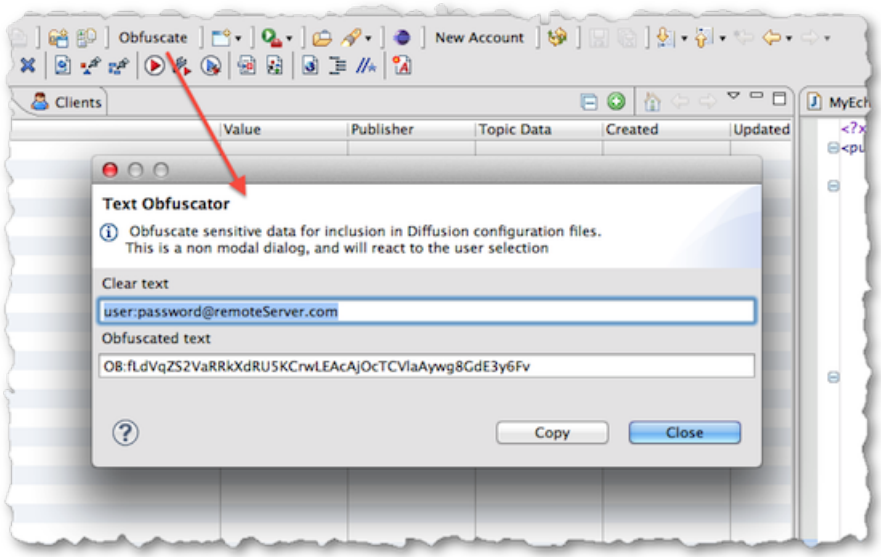


Figure 90: Property Obfuscator Dialog

The dialog is modeless, and reacts to the users selection changes so it need not be dismissed if there is more than one value to obfuscate.

Copyright 2013 Push Technology

Copyright 2013 Push Technology

Chapter 23

Demos

Topics:

- *Demos*

Diffusion comes with examples / demos that demonstrate certain features of Diffusion. The examples, which are included as source in the installation, are also compiled in the Diffusion.jar.

Demos

Diffusion™ comes with examples / demos that demonstrate certain features of Diffusion™.

If you have chosen to include Demos as part of the installation process, you will find a collection of individual DAR files in the `demos` folder within your Diffusion root directory, as well as a `Demos.jar` file which contains all of the demos for distribution. To enable a demo to run, copy its DAR file into the `/deploy` folder at the root of your Diffusion installation, and start Diffusion.

The examples are included as source in the installation.

The demos in the **Demos** section of the default Diffusion installation webpage demonstrate various applications of Diffusion™. They also contain a connection widget, allowing you to modify the transports used as well as inspect messages incoming and outgoing from the Client.

To access the Diffusion webpage and demos, start the web server and then type in the following url into a browser.
http://localhost:8080

Flex Trade Demo	This is a basic multi asset trading platform demo which includes video, chat, news and a deal blotter.
Tech Magnets	This demo is about client modified state model. The tiles are stored in state in Diffusion™ Server. When a tile is moved by the client, a message is sent to the Diffusion™ Server, the state is modified and an Exclusive message is sent to all other subscribers. The reason for this is that the client that moved the tile, doesn't need to know that the state has changed.
Chat Demo	This is a simple demo that uses the allows for messages from a client to be broadcast to other clients. The publisher also uses **Topic Loaders** and **Client Listeners**
Dogfight (game) demo	Aerial combat game for up to 8 players. Use the arrow keys to control your plane and the space key to fire. Uses HTML 5 features.
Drawingboard Demo	This is a demo that demonstrates the use of real-time interaction. Due to the amount of data required, Fragmented messages are used to ensure performance is stable for all clients.

Copyright 2013 Push Technology

Chapter 24

Testing

Topics:

- *Flex/Flash Client*
- *Java Client*
- *Java Event Publisher Test Tool*
- *Javascript Client Test Tool*
- *Silverlight Client Test Tool*
- *Stress Test Tuning*
- *Stress Test*
- *Test Tools*
- *Windows Client Test Tool (.NET)*
- *Windows Event Publisher Test Tool (.NET)*

This section covers some aspects of testing a Diffusion™ system.

Test Tools	Tools available to aid with testing.
Stress Test	Tools available to check performance testing.

Flex/Flash Client

The Flex Client test tool is a Flex application that uses the DiffusionTransport.swc library. The tool can be found in the Tools section in the Root Menu of your Diffusion™ installation.

This tool enables the user to supply connection information as well as credential information. Select which transport is going to be used to connect to the Diffusion™ Server. It is also possible to issue a server ping from this client. The ping response will then display the elapsed time, its average and the current queue size for this client.

This tool can also be used to test Reconnection, Auto Failover and Load Balancing. Available servers can be listed by IP in the URL box. The 'Reconnect' button will, after a 'Disconnect', return the client to the server it was originally connected to, and receive messages queued during disconnection. Checking 'Auto Failover' will cause the client to connect to the next available server, after the existing connection is forcibly killed. Checking 'Cascade' will cause the client to cascade through the available servers until it finds one to which it can connect. Check 'Load Balancing', in conjunction with Auto Failover, and the client will connect to one of a 'shuffled' list of available servers.

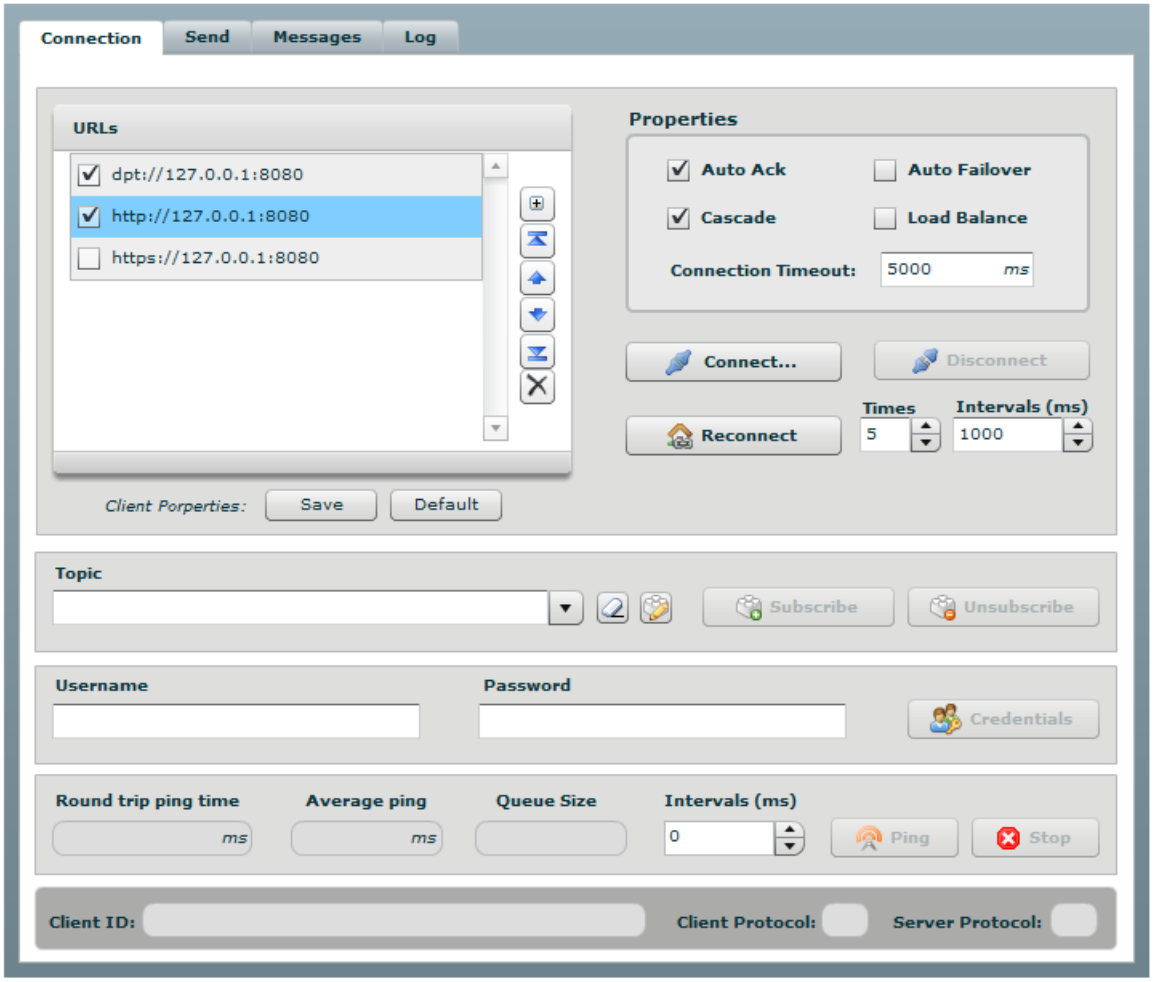


Figure 91: Flex Client

It is now possible to pass user headers with a 'fetch' request which will be reflected back to the client in the response message as a Correlation ID. This allows the client to be able to associate replies with requests. The Flex Client API thus has a new 'headers' parameter on the fetch methods. No changes are required to Publishers in order to benefit from this.

Copyright 2013 Push Technology

Once connected to a Diffusion™ server it is then possible to send messages to any topic. The publisher will then receive this message and notify the

```
messageFromClient
```

method in the publisher. The requested encoding can be set, as well as any user headers required. It is also possible to request an acknowledgement from the server.

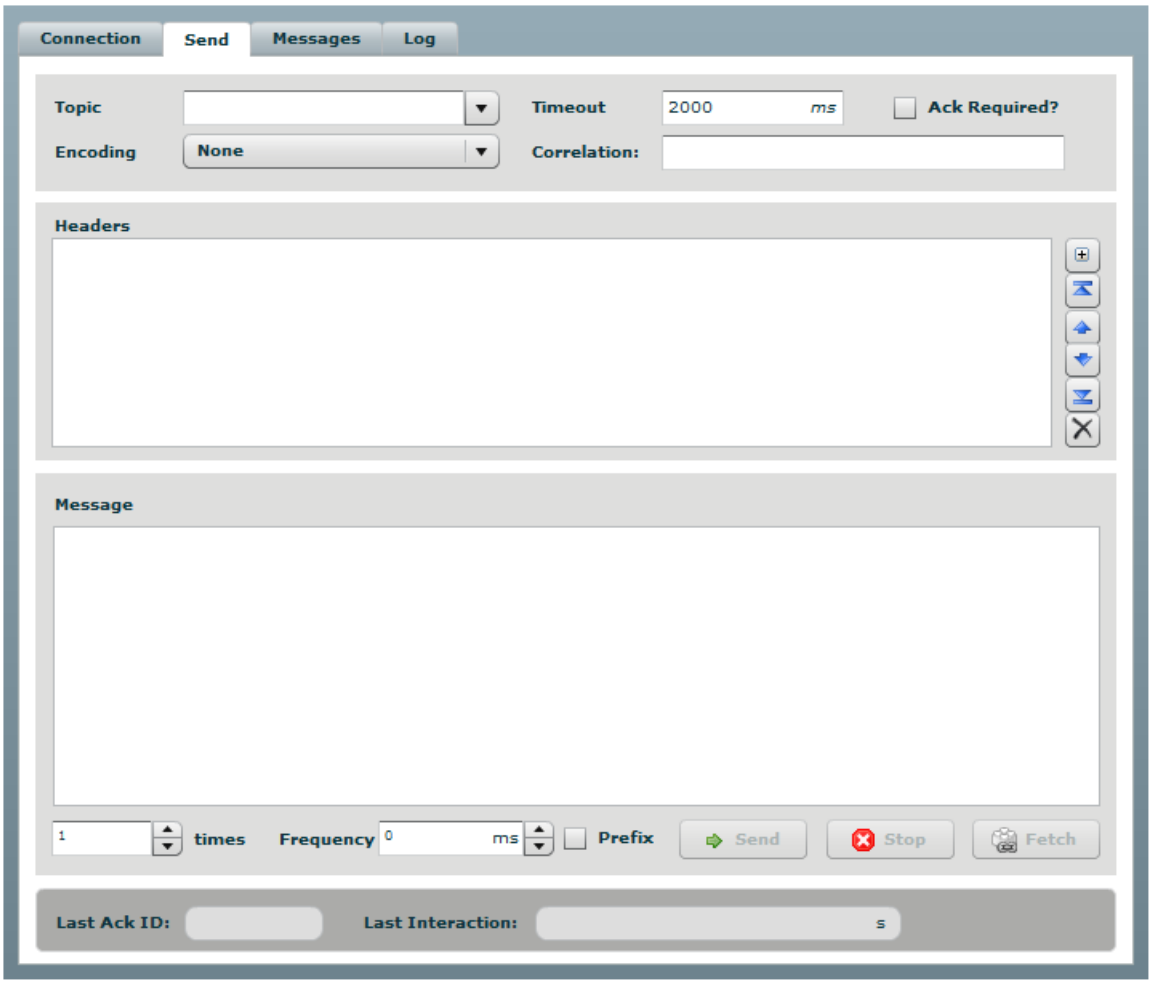


Figure 92: Sending Messages

Messages are displayed in a tree format. It is not a true hierarchic tree, as each topic has its own node directly off the root node.

Copyright 2013 Push Technology

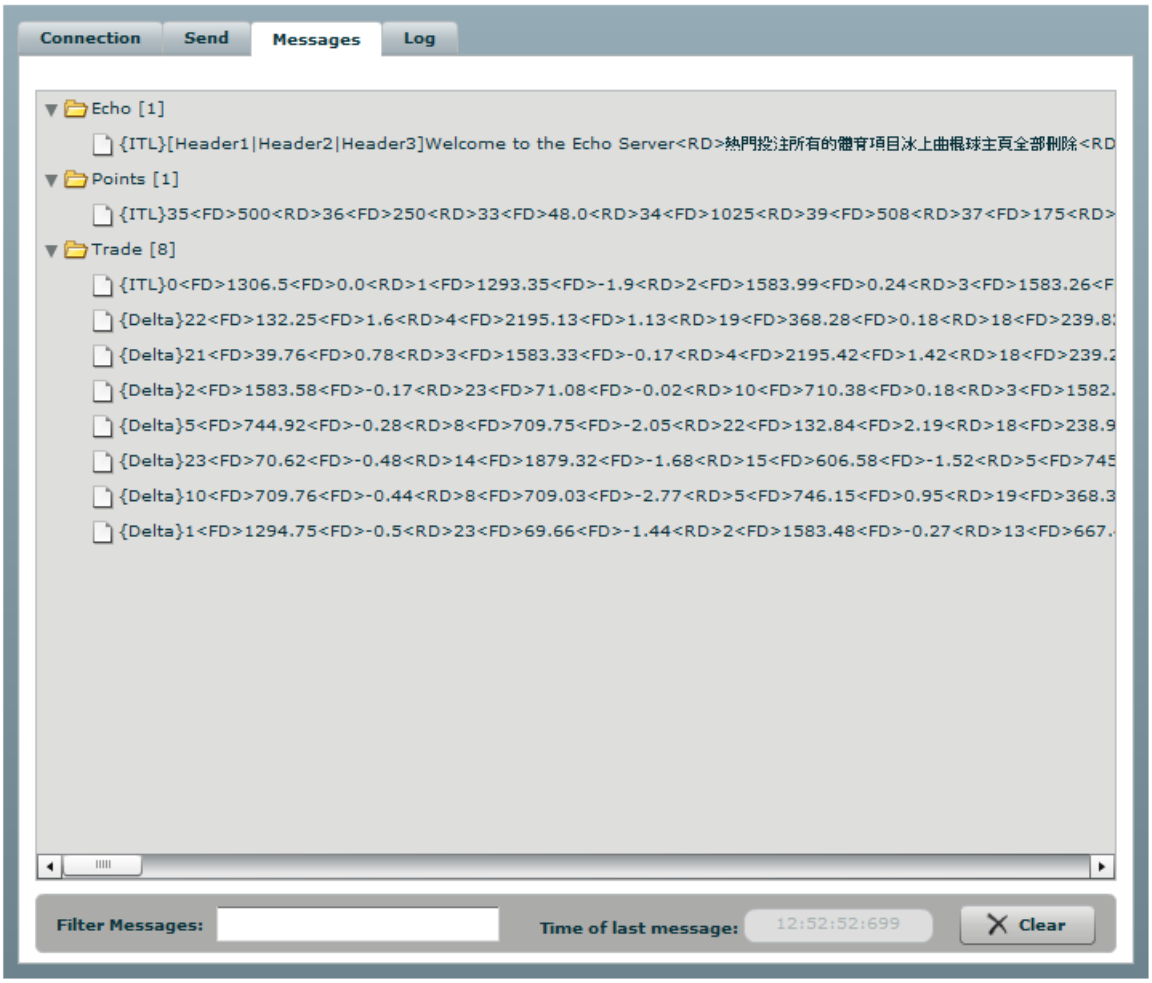
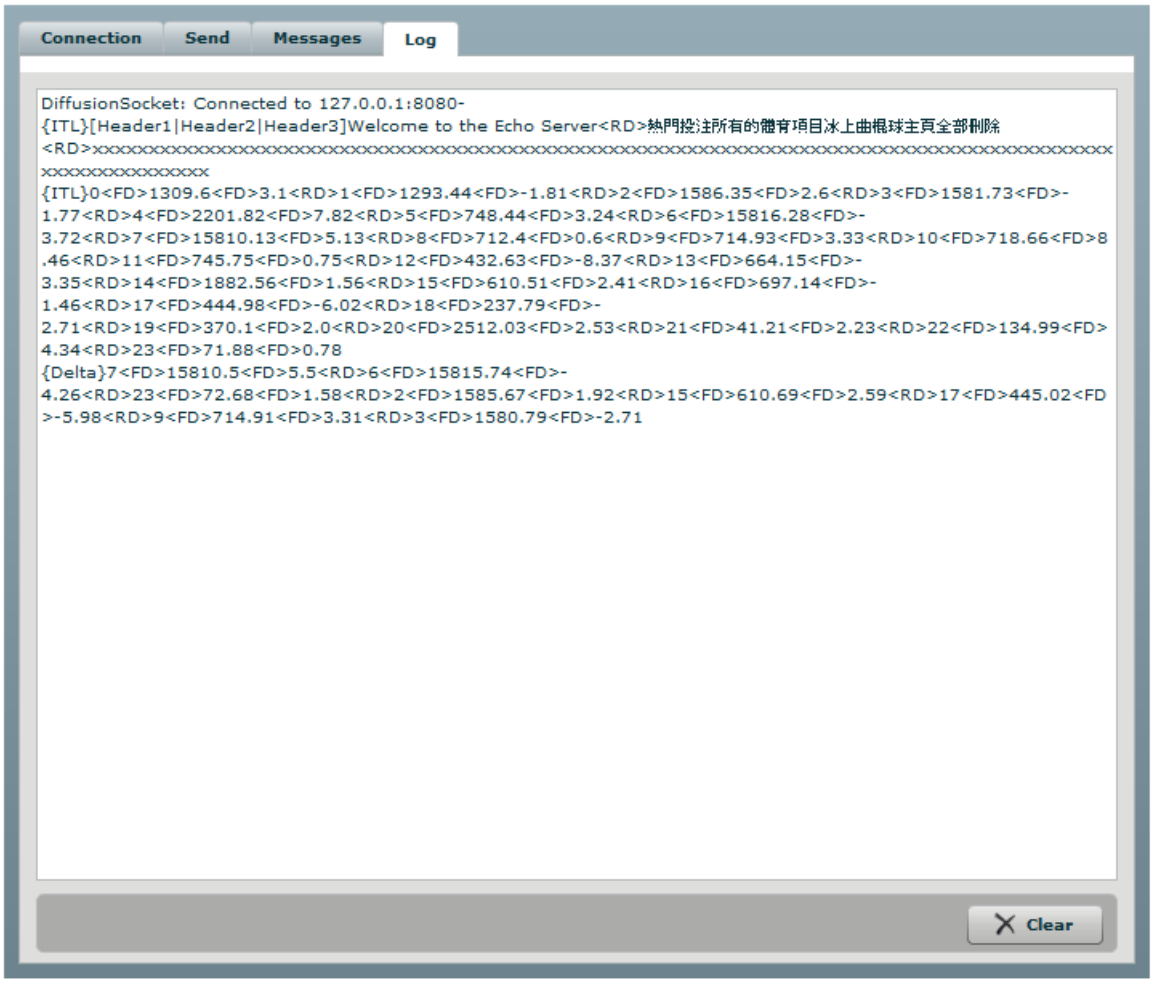


Figure 93: Message tree displayed

The log tab shows messages that are sent from the Diffusion™ Client when debugging is switched on.

Copyright 2013 Push Technology


Connection
Send
Messages
Log
DiffusionSocket: Connected to 127.0.0.1:8080-
{ITL}[Header1|Header2|Header3]Welcome to the Echo Server<RD>熱門投注所有的體育項目冰上曲棍球主頁全部刪除
<RD>xxxx

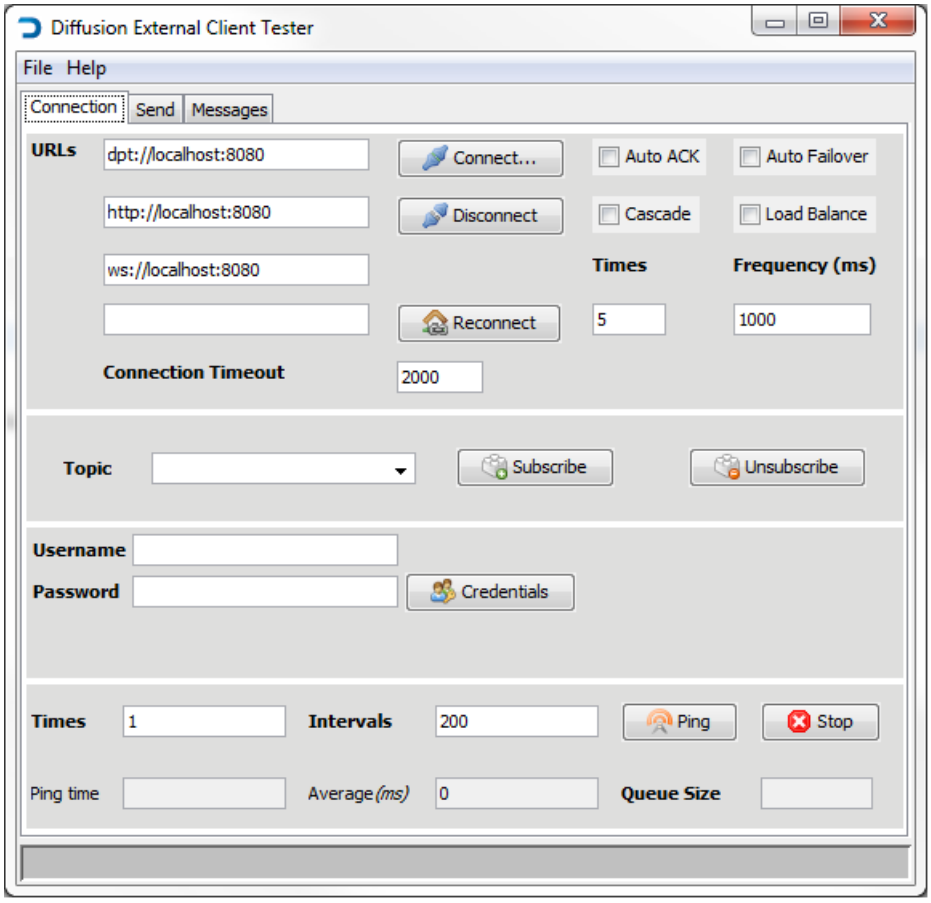


Figure 95: External Client tester

It is also possible to connect to Diffusion™ securely. In order for the external client to work in secure mode, a certificate needs to be installed on the client machine. To aid in installing the certificate, there is an option under the file menu called Load Cert. This will connect to Diffusion™ and detail instructions on how to install the newly created jssecerts file.

Once subscribed to a topic it is then possible to send messages to that topic. The publisher will then receive this message and notify the

```
messageFromClient
```

method in the publisher. It is also possible to set the message encoding type and any user headers if required.

Copyright 2013 Push Technology

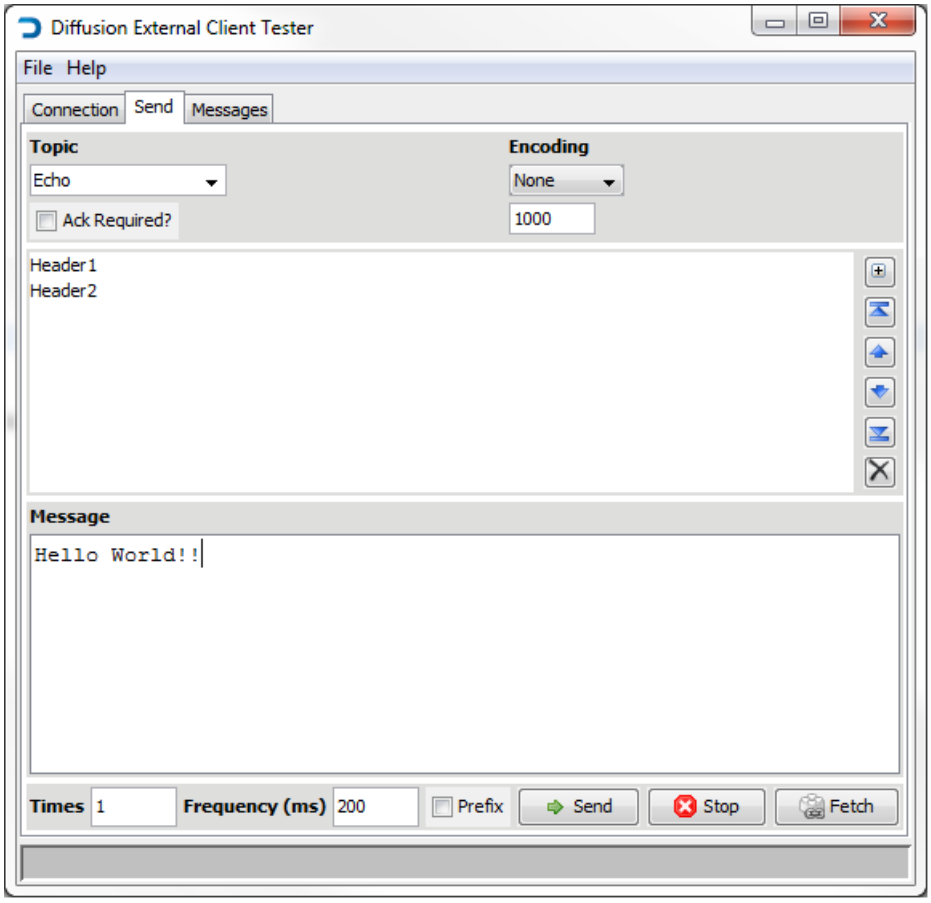


Figure 96: Sending messages

Messages are displayed in a tree format. It is not a true hierarchic tree, as each topic has its own node directly off the root node.

Copyright 2013 Push Technology

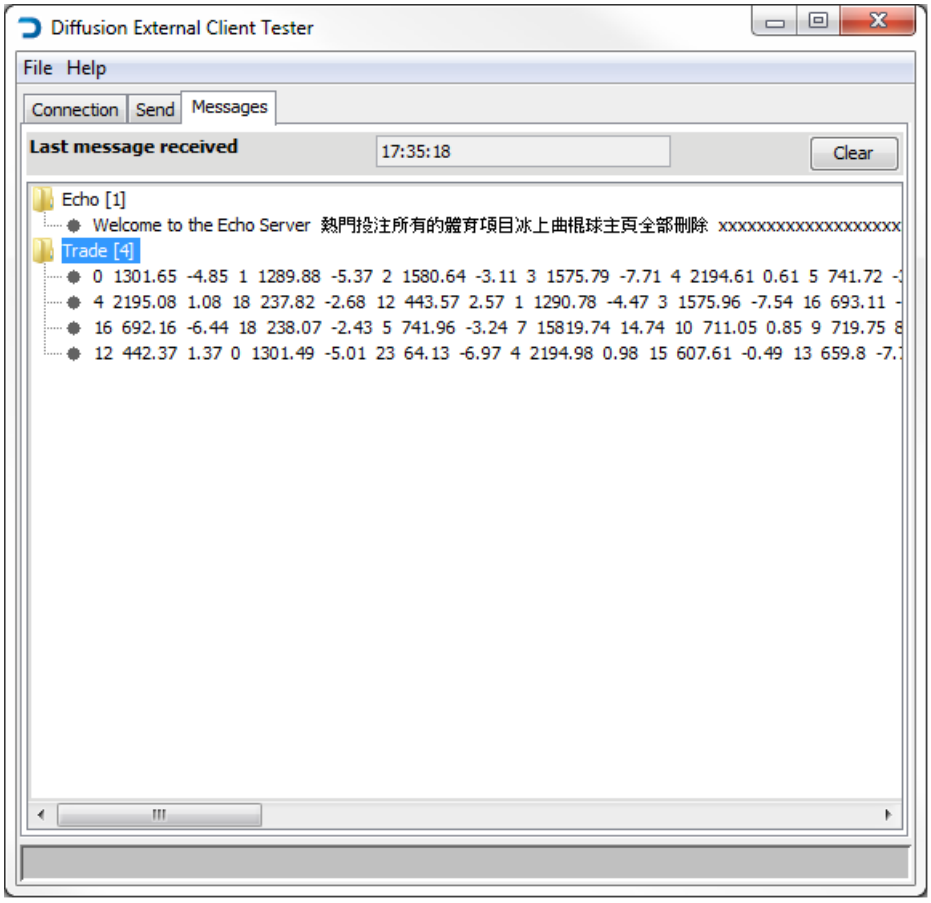


Figure 97: Messages displayed

It is also possible to examine a message by double clicking it

Copyright 2013 Push Technology

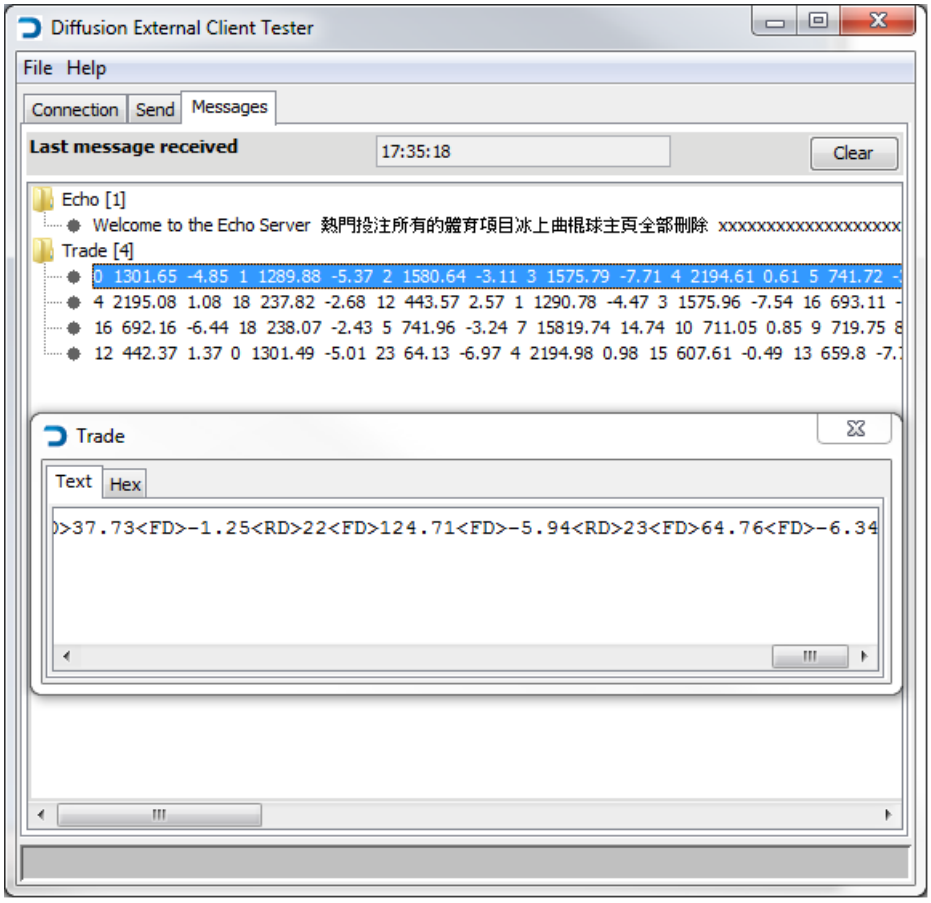


Figure 98: Examining a message

Java Event Publisher Test Tool

The event publisher test tool is found in the Tools folder of your Diffusion™ installation, and is run from the command line using either eventpub.sh or eventpub.bat. It enables messages to be sent to a Diffusion™ sever. This basically enables the user to send messages to the publisher that is managing the topic specified.

Copyright 2013 Push Technology

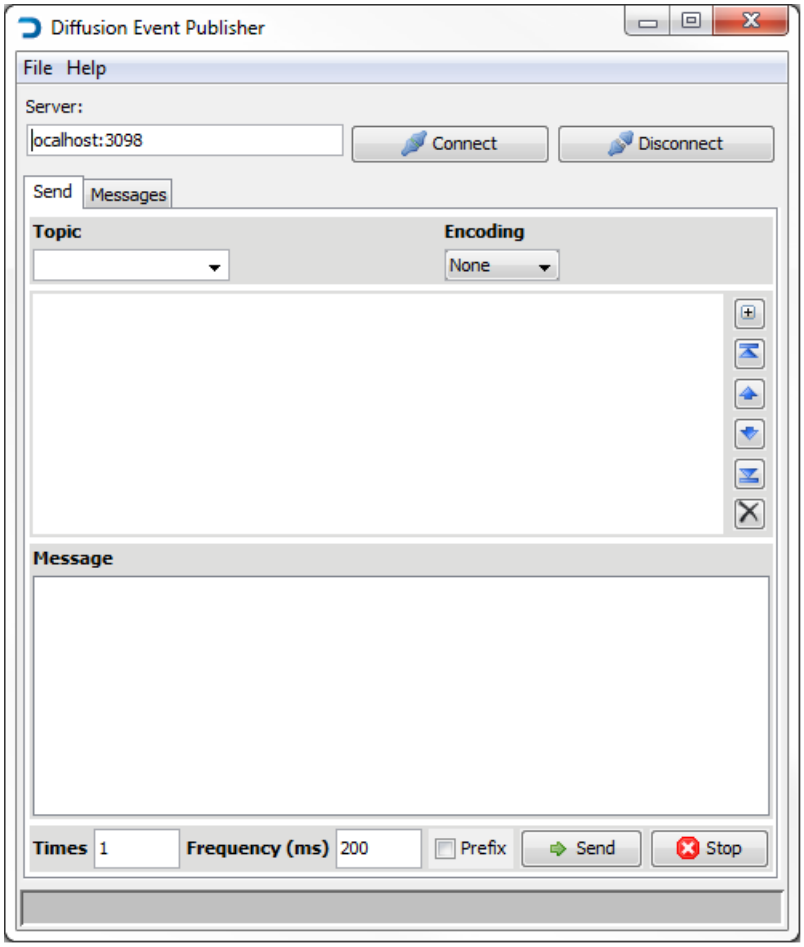


Figure 99: Event Publisher Test Tool

The publisher will get notified of the message on the `messageFromEventPublisher` method.

Copyright 2013 Push Technology

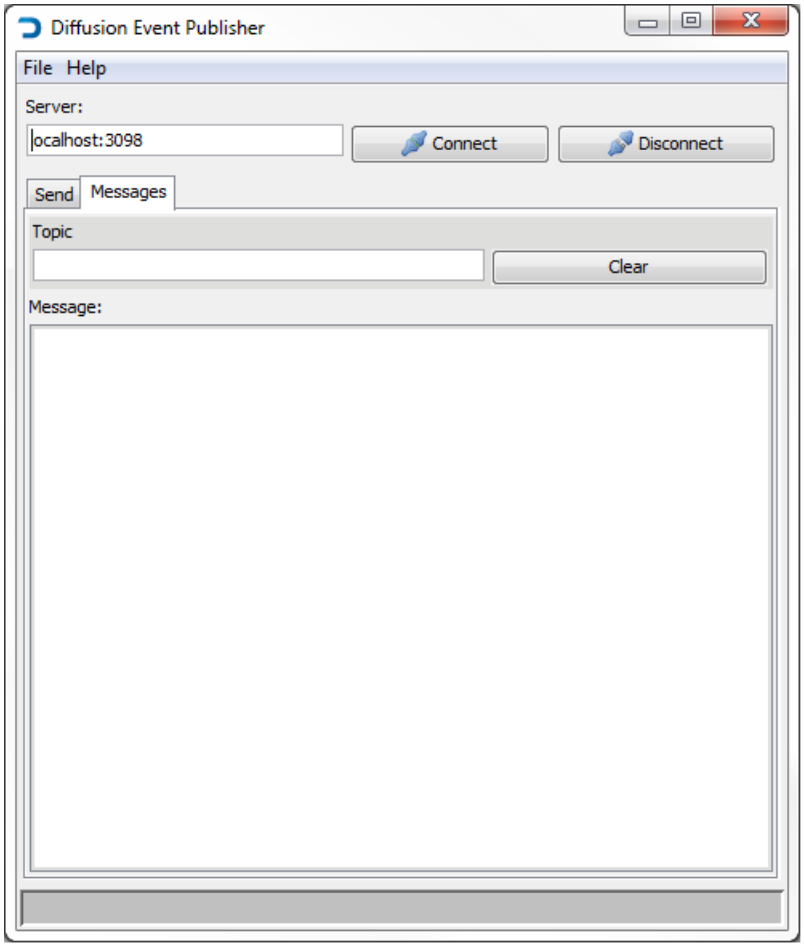


Figure 100: Sending messages

Javascript Client Test Tool

The Javascript test tool allows the user to connect to a Diffusion™ publisher and test the API.

The Javascript test tool is browser based and can be found in the Root Menu of your Diffusion™ installation. It allows for different types of connections. The test tool also allows for Subscribing, Un-subscribing and Fetching against topic sets.

Fetch Correlation allows user headers to be passed with a 'fetch' request which will be reflected back to the client in the response message. This allows the client to be able to associate replies with requests. The Javascript Client API thus has a new 'headers' parameter on the fetch methods. See Javadoc for changes to the API specification. No changes are required to Publishers in order to benefit from this.

Copyright 2013 Push Technology

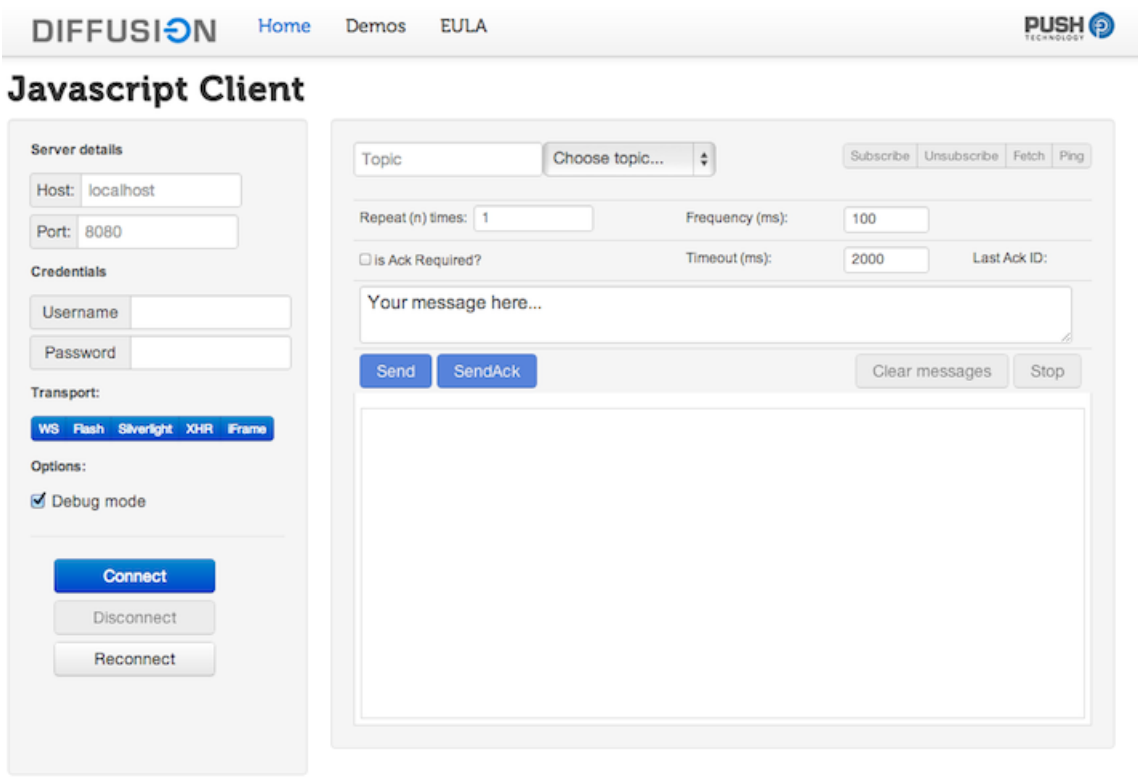


Figure 101: Javascript test tool

Silverlight Client Test Tool

The Silverlight Client test tool is a Silverlight application that uses the `PushTechnology.Transports` assembly. This tool enables the user to perform various diagnostic functions without having to build a new Silverlight application.

Copyright 2013 Push Technology

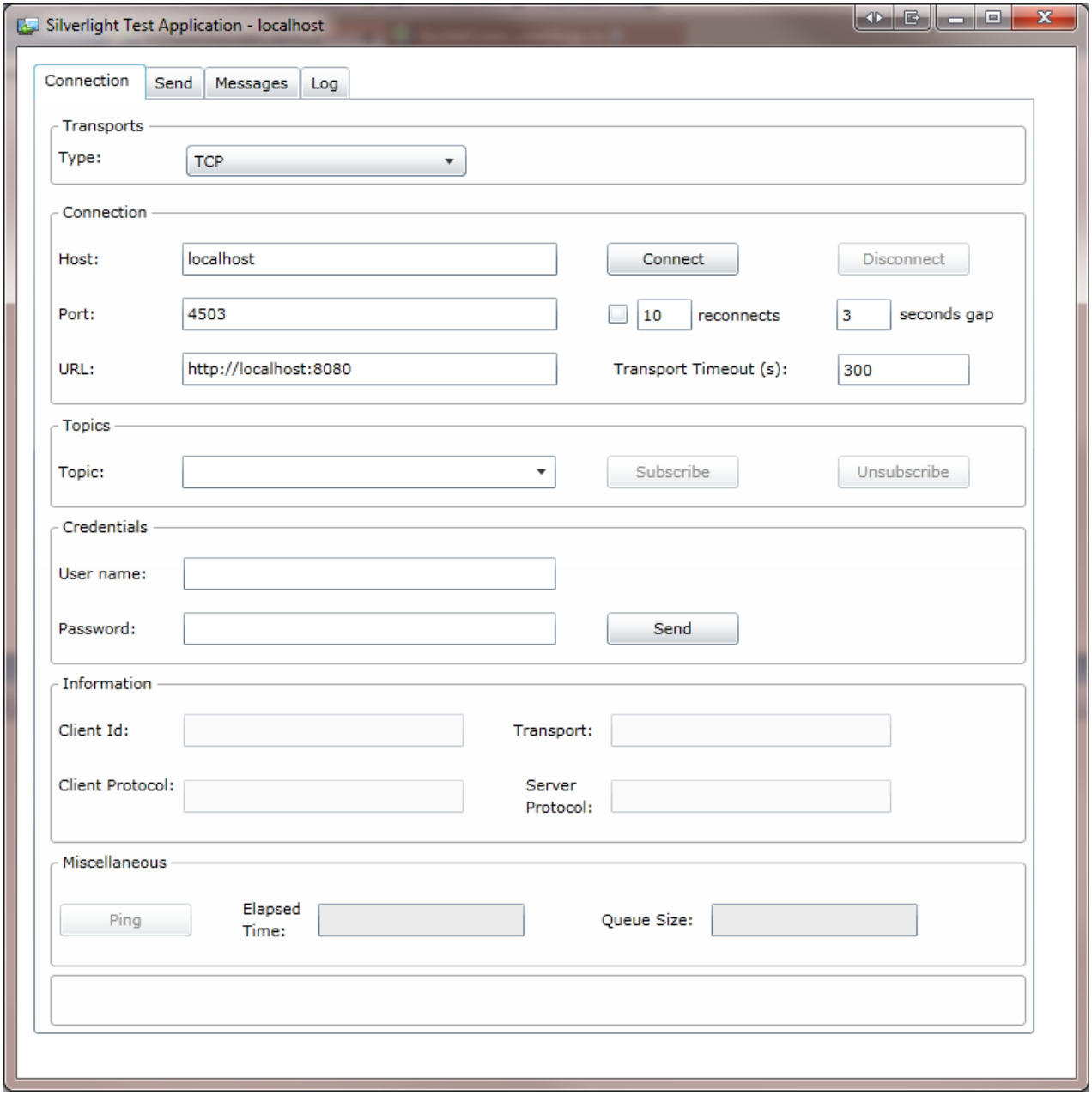


Figure 102: Silverlight test tool

Once connected to a Diffusion™ server it is then possible to send messages to any subscribed topic. The Diffusion™ Publisher will then receive this message and notify the messageFromClient method in the Publisher. The requested encoding can be set, as well as any user headers.

Copyright 2013 Push Technology

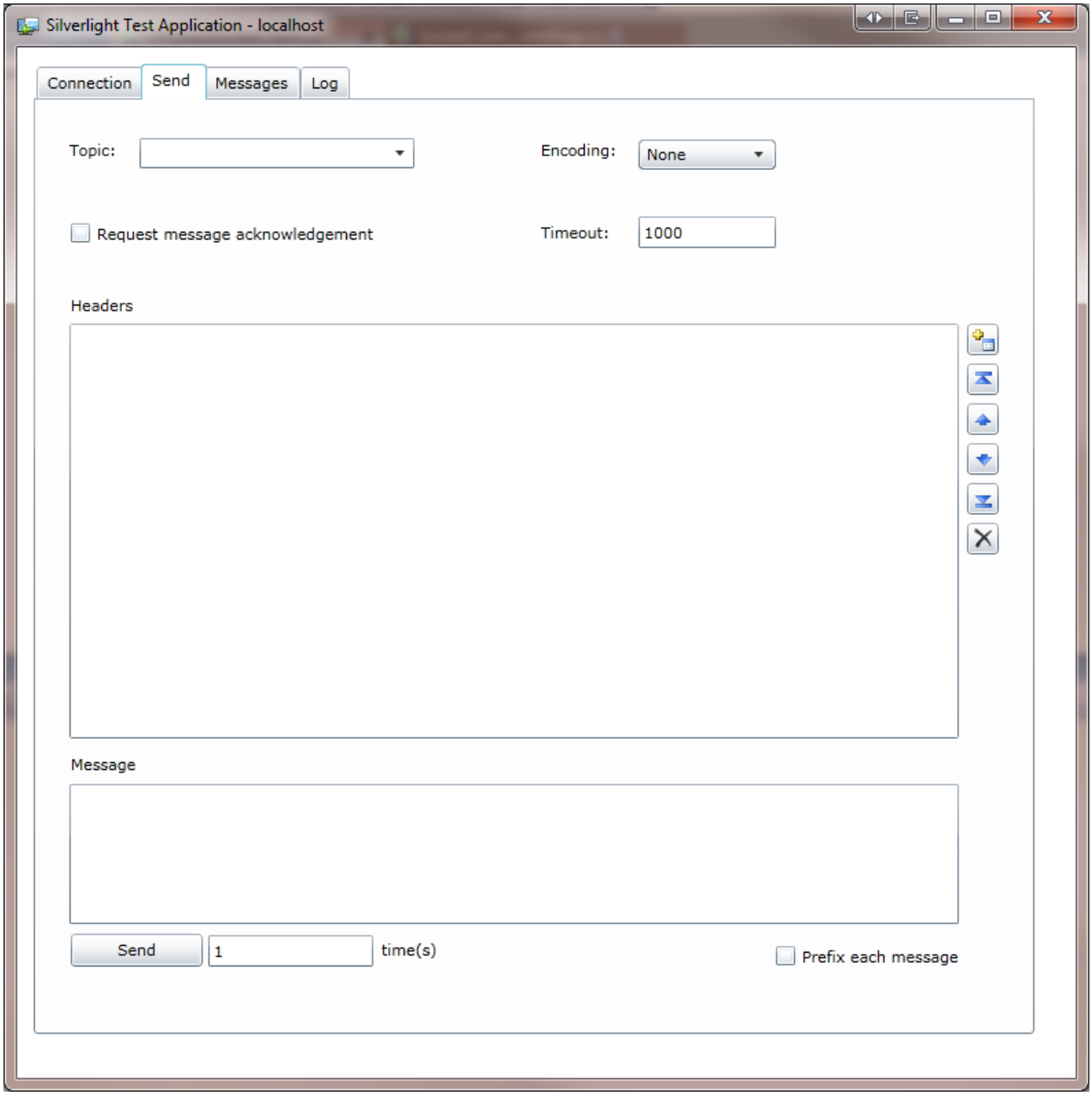


Figure 103: Sending messages

Received messages are displayed in a hierarchical format:

Copyright 2013 Push Technology

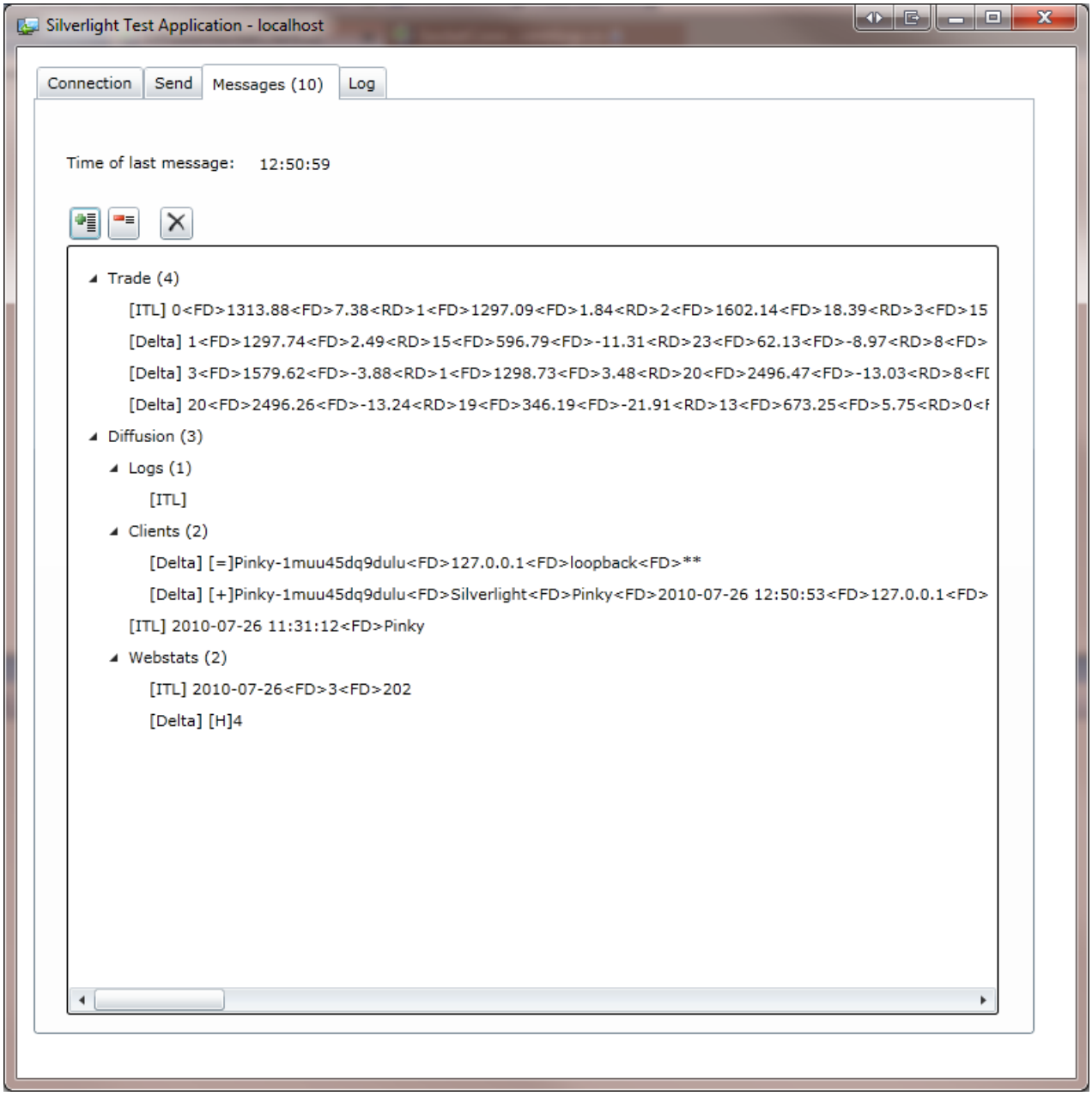


Figure 104: Displayed messages

Stress Test Tuning

Stress testing Diffusion™ will require various tuning changes to be made:- See the tuning section for more in depth information

Increase Number of Open Files	If the Diffusion™ Server is running in Linux then there is a limit to the number of open files/connections. See "Increasing Number of Open Files".
Increase Number of Acceptors	Performance gains can be obtained by increasing the number of acceptors. (The number of physical acceptors that handles incoming requests).

Copyright 2013 Push Technology

	For Flash clients, the value in *etc/Connectors.xml* needs increasing. These should be increased carefully and the Server re-tested each time. Depending on the tests there will come a point where the increase has a detrimental effect. It is advisable to make small increases then re-test (e.g. 5-10).
Buffer Sizes	Socket Buffer Sizes can be changed in *etc/Connectors.xml*
Inbound Thread settings	The Inbound Thread Pool settings will need tuning otherwise the Pool and Queues will get full. The settings are found in *etc/Server.xml* . They should be increased carefully and the server re-tested each time. The maximum size and queue size should be set quite high for stress testing (e.g. 250) so that the server can handle the hundreds or thousands of simultaneous connection attempts.
Client Queues	Depending on the number and frequency of the messages, the Client Queue parameter may need to be increased.
Client Multiplexers	By default, there are two Client Multiplexers configured. This may need to be increased as the client load on each Diffusion™ Server increases.

Custom Stress Test

The supplied Stress Test package looks at the *etc/Publishers.xml* and stresstest.properties for its configuration. As the source of the stress test package is supplied there is nothing to stop the user extending the supplied Stress Test package as a framework for their own custom tests. It is worth Stress Testing the publisher that you wish to go live with as this is going to show any performance degradation under duress, and will also highlight any data synchronicity issues. It might also be worth changing the default behaviour of the clients as well the the client connection ratio that you expect. For instance there would be no point in testing HTTP Clients, if your solution is going to be Flash only, but if your solution is going to be DHMTL, then it is worth setting up a ratio of plug-in (flash) clients and HTTP clients. If your solution is going to be solely SSL, then it is only worth configuring the SSL clients.

Limitations

If more than 2000 clients are to be simulated then it is better to run the stress test from multiple machines. Most machines will start to struggle when dealing with 2000 concurrent connections. By running the stress test on multiple machines any number of clients can be tested.

Socket exceptions can be encountered when a large number of clients are run on one machine. These can be thrown on either the Diffusion™ Server or within the stress test tool. If this is occurring then the number of Clients needs to be reduced

Stress Test

Stress test is a good way of working out what the system requirements are going to be of a Diffusion™ Server under duress. The Stress test tool is issued to test the performance of Diffusion™ under stress and helps to understand how changes to the configuration can have an effect upon the performance.

Copyright 2013 Push Technology

The stress test tool should not be run on the same machine as the Diffusion™ Server. A separate machine is therefore required to run the stress test tool.

Installing the Stress Test Tool;

1. On the test machine, create a `stresstest` directory and copy the files `stresstest/stresstest.bat`, `stresstest/stresstest.sh`, `stresstest/stresstest.properties` and `lib/diffusion.jar` from the Diffusion installation
2. Configure Stress Test by editing the `stresstest.properties` file. The following need to be configured:
3. Number Of Clients
4. Transport type
5. Server details
6. Number of Messages with which to run the test
7. The Topic to which the Clients will subscribe
8. On a Windows machine launch `stresstest.bat`, on a Unix machine launch `stresstest.sh`.

Test Tools

Test Tools are provided which allow you to connect to a Diffusion™ Server as an External Client or Event Publisher Application. There are also tools that act as a Publisher Server application.

Testing Tools

Generic Java versions are provided which work on any platform and Windows versions are provided specifically for use on Windows platforms.

Java Client	Provides a GUI interface simulating an External Client Connection using the Java API. Suitable for use on any platform.
Java Event Publisher	Provides a GUI interface simulating an Event Publisher Connection using the Java API. Suitable for use on any platform.
Windows Client	Provides a GUI interface simulating an External Client Connection using the Windows API.> Suitable for use on Windows platforms only.
Windows Event Publisher	Provides a GUI interface simulating an Event Publisher Connection using the Windows API. Suitable for use on Windows platforms only.
JavaScript Client	Provides a GUI interface simulating a JavaScript Client Connection. Suitable for use on any platform.
Flex/Flash Client	Provides a GUI interface simulating a Flash Client Connection. Suitable for use on any platform where Flash Player is installed.
Silverlight Client	Provides a GUI interface simulating a Silverlight Client Connection. Suitable for use on any platform where Silverlight is installed.

Copyright 2013 Push Technology

Windows Client Test Tool (.NET)

The External Client Test Tool is a .NET application which will enable the user to connect/subscribe/unsubscribe to a Diffusion™ server. It allows for many different connection types (eg HTTP/DPTS), and also allows for the composition and sending of messages.

Below is the 'Connection' tab of the External Client Test Tool; this contains all of the facilities required to establish a connection to the Diffusion™ server.

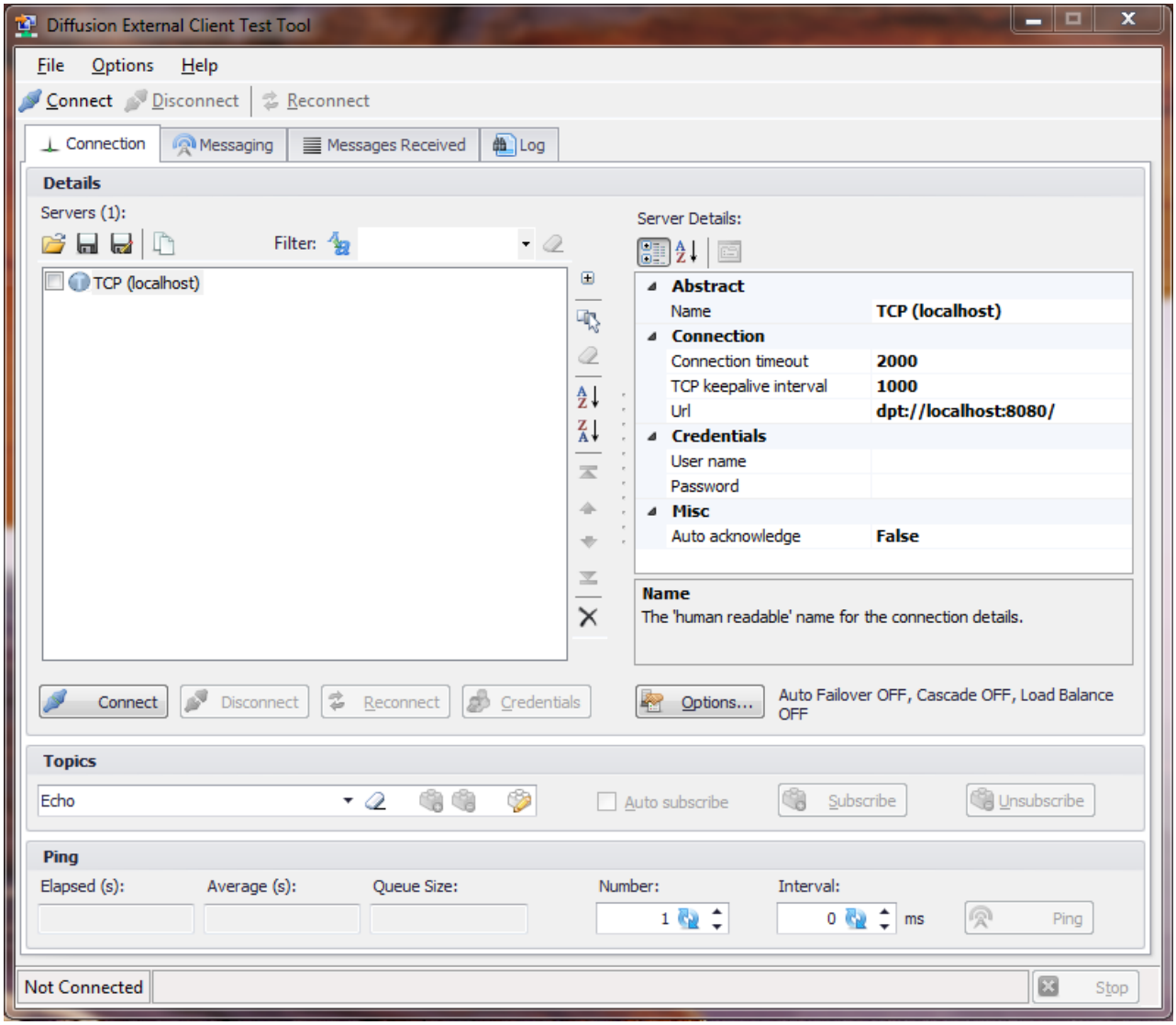


Figure 105: External Client test tool

Additional topics can be subscribed at any time subsequent to connection by entering the topic names in the Topics field and clicking the 'Subscribe' button. The converse action (i.e. unsubscription) can be performed with the 'Unsubscribe' button.

It is also possible to issue a server ping from this client; the elapsed time will be indicated in the relevant fields, along with a rolling average.

If connection credentials are required in order to subscribe to a particular topic, these can be entered in the 'Credentials' section. They can also be sent retrospectively using the 'Credentials' button.

Various connection options are available to the user via the 'Options' button:

Copyright 2013 Push Technology

Auto Failover

Once a connection to Diffusion™ is placed, should it fail, and if there is a pool of server details to pick from, then the next set of server details is chosen and connection placed to it.

Cascade

When the client software attempts to place a connection, should the attempt fail, then the next set of server details in the list is chosen and used. It is similar to Auto Failover except this logic is applied prior to a connection, whereas Auto Failover is applied once a connection is in place.

Load Balance

If 'Load Balance' is set, the list of server details is shuffled prior to use.

Restore state when reconnecting

If connection to Diffusion™ is lost, and the server has been configured accordingly, the client will reconnect and receive any messages that have been 'missed' when the connection failed.

Once subscribed to a topic, it is then possible to send messages to either that specific topic, or any registered topic, using the 'Send' tab:

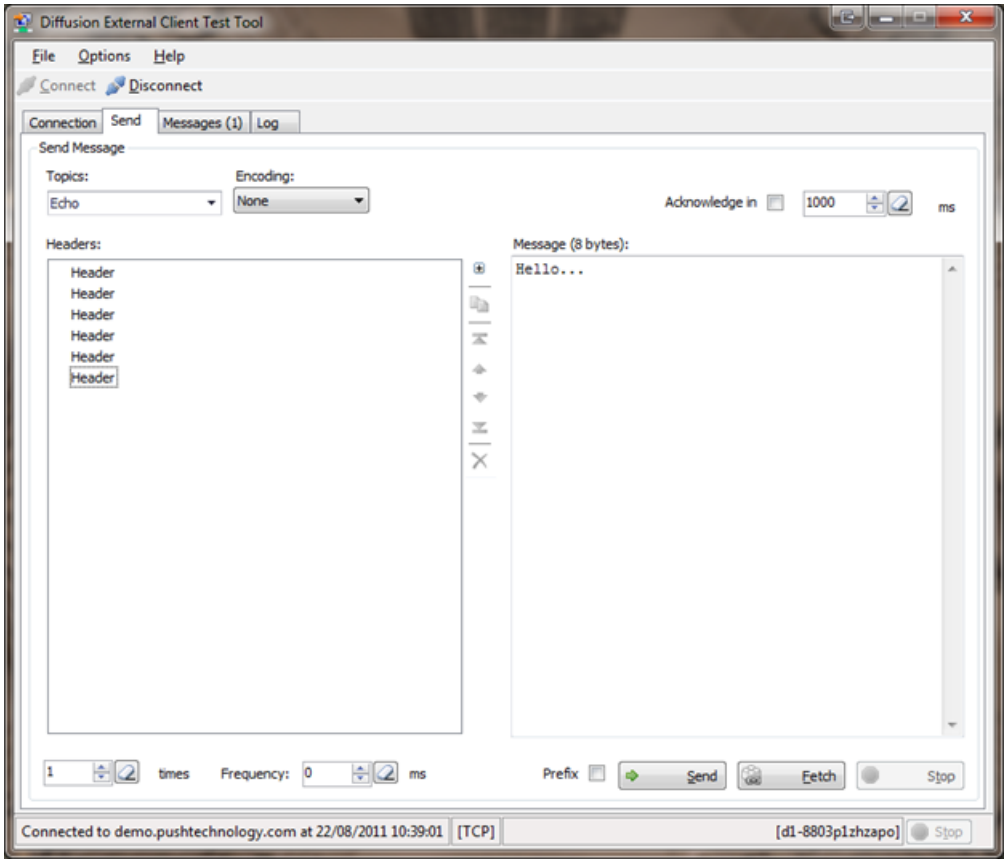


Figure 106: Sending messages

As can also be seen, it is possible to compose a message containing headers; these can be added using the convenient toolbar to the side of the main 'headers' area. It is also possible to send a message multiple times by incrementing the edit field at the bottom of the message composition area.

Any received messages from the Diffusion™ server are shown in the 'Messages' tab thus:

Copyright 2013 Push Technology

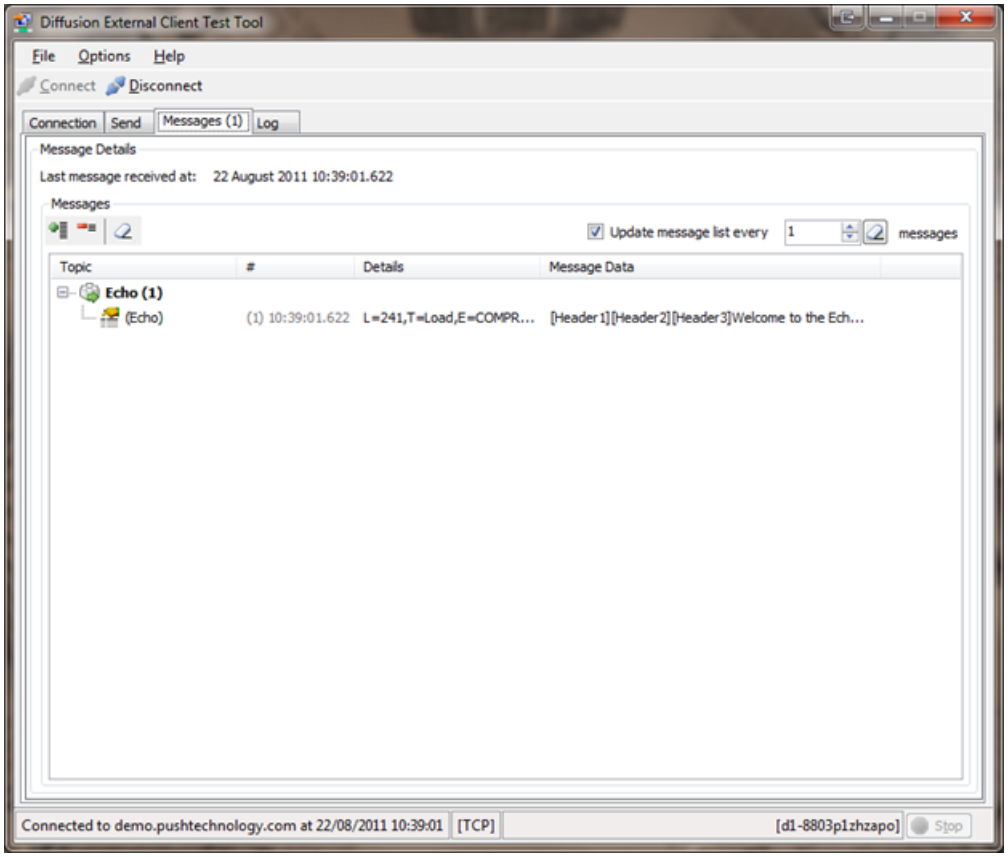


Figure 107: Displayed messages

Messages are displayed in a true hierarchical format, so deeply-nested topics will be relatively easy to locate and analyse.

If a high volume of messages is expected from the Diffusion™ server, it is possible to filter the number of messages by adjusting the 'Update message list' value. It is also possible to disable message updates completely by clicking the check box.

It is also possible to view logging information generated by the .NET libraries:

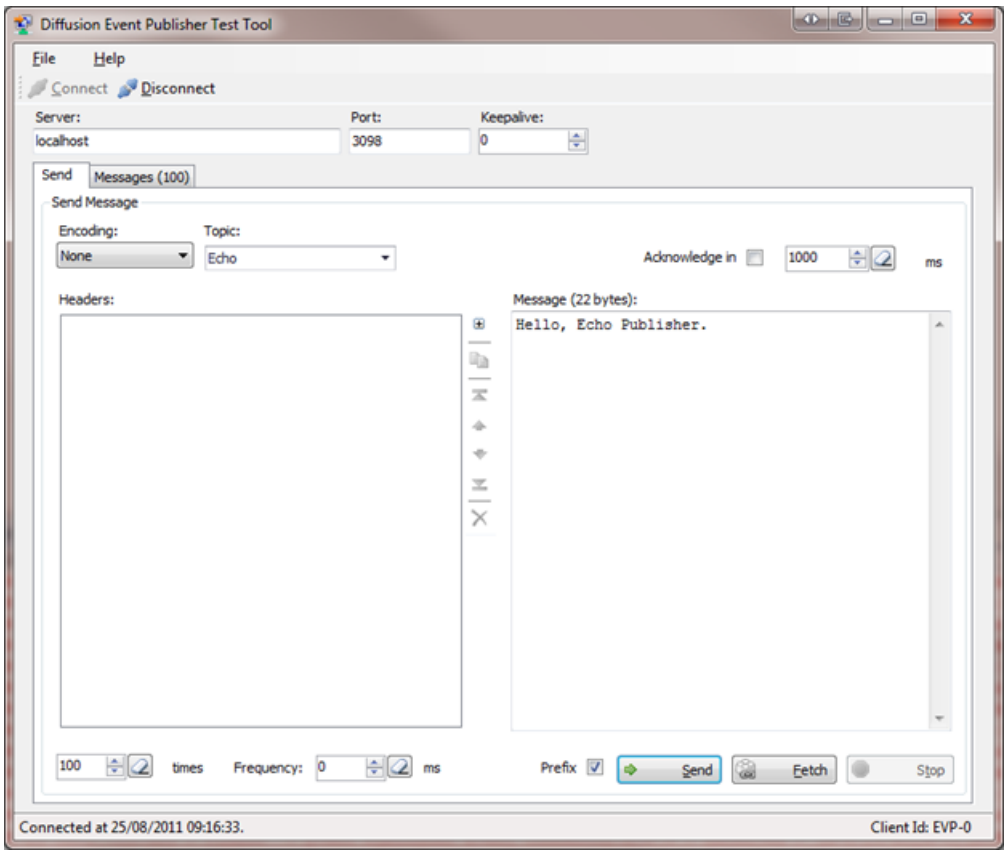
The logging type and level can be adjusted using the 'Preferences' option on the main menu:

Windows Event Publisher Test Tool (.NET)

The event publisher test tool enables messages to be sent to a Diffusion™ server. This basically enables the user to send messages to the publisher that is managing the topic specified.

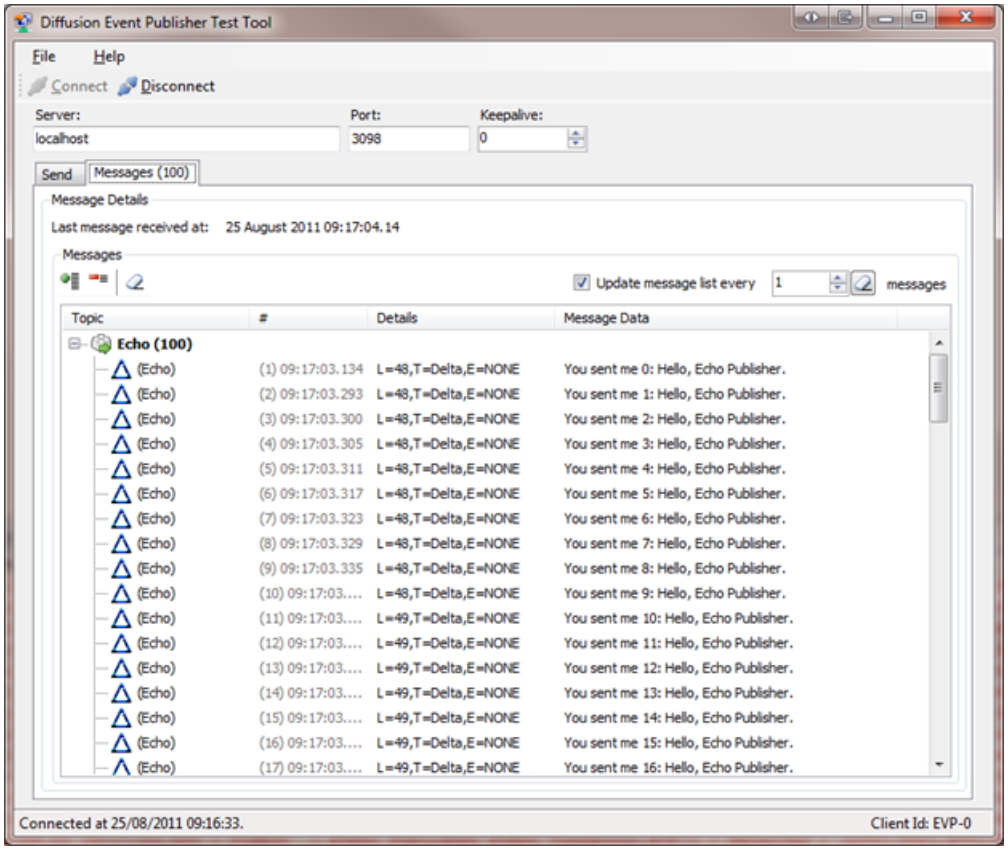
The event publisher test tool enables messages to be sent to a Diffusion™ server. This enables the user to send messages to the publisher that is managing the topic specified.

Copyright 2013 Push Technology



As an Event Publisher is bi-directional, any messages received are displayed in the 'Messages' tab:

Copyright 2013 Push Technology


Diffusion Event Publisher Test Tool
File
Help
Connect
Disconnect
Server:
localhost
Port:
3098
Keepalive:
0
Send
Messages (100)
Message Details
Last message received at: 25 August 2011 09:17:04.14
Messages
Update message list every
1
messages
Topic
#
Details
Message Data
Echo (100)
(Echo) (1) 09:17:03.134 L=48,T=Delta,E=NONE You sent me 0: Hello, Echo Publisher.
(Echo) (2) 09:17:03.293 L=48,T=Delta,E=NONE You sent me 1: Hello, Echo Publisher.
(Echo) (3) 09:17:03.300 L=48,T=Delta,E=NONE You sent me 2: Hello, Echo Publisher.
(Echo) (4) 09:17:03.305 L=48,T=Delta,E=NONE You sent me 3: Hello, Echo Publisher.
(Echo) (5) 09:17:03.311 L=48,T=Delta,E=NONE You sent me 4: Hello, Echo Publisher.
(Echo) (6) 09:17:03.317 L=48,T=Delta,E=NONE You sent me 5: Hello, Echo Publisher.
(Echo) (7) 09:17:03.323 L=48,T=Delta,E=NONE You sent me 6: Hello, Echo Publisher.
(Echo) (8) 09:17:03.329 L=48,T=Delta,E=NONE You sent me 7: Hello, Echo Publisher.
(Echo) (9) 09:17:03.335 L=48,T=Delta,E=NONE You sent me 8: Hello, Echo Publisher.
(Echo) (10) 09:17:03.... L=48,T=Delta,E=NONE You sent me 9: Hello, Echo Publisher.
(Echo) (11) 09:17:03.... L=49,T=Delta,E=NONE You sent me 10: Hello, Echo Publisher.
(Echo) (12) 09:17:03.... L=49,T=Delta,E=NONE You sent me 11: Hello, Echo Publisher.
(Echo) (13) 09:17:03.... L=49,T=Delta,E=NONE You sent me 12: Hello, Echo Publisher.
(Echo) (14) 09:17:03.... L=49,T=Delta,E=NONE You sent me 13: Hello, Echo Publisher.
(Echo) (15) 09:17:03.... L=49,T=Delta,E=NONE You sent me 14: Hello, Echo Publisher.
(Echo) (16) 09:17:03.... L=49,T=Delta,E=NONE You sent me 15: Hello, Echo Publisher.
(Echo) (17) 09:17:03.... L=49,T=Delta,E=NONE You sent me 16: Hello, Echo Publisher.
Connected at 25/08/2011 09:16:33.
Client Id: EVP-0

Copyright 2013 Push Technology

Chapter 25

Tools

Topics:

- *Tools*
- *Tools for Amazon EC2*
- *Tools for Joyent*

If the tools were installed during the installation process, there are some tools that can help with the monitoring of Diffusion™ plus some handy utilities.

Tools

Property Obfuscator

The property obfuscator is deprecated since v4.6 and is being replaced by *equivalent functionality in the Introspector* .

This tool can be used to hide any Diffusion™ configuration file entries that are sensitive, e.g. a password. Enter the text that you want to obfuscate and press the button. All obfuscated entries will always start with `OB:`

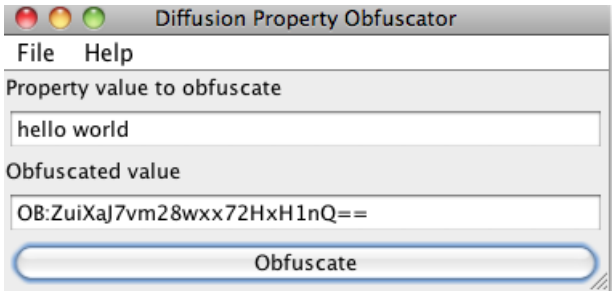


Figure 108: Property Obfuscator

This can also be run in a headless environment from the command line supplying the first argument as the string that you want to Obfuscate.

Tools for Amazon EC2

This is a description of a number of tools and adaptations which are provided for use when using Diffusion™ with the Amazon Elastic Compute Cloud (EC2).

Diffusion includes in `tools/ec2/` a number of files useful when deploying Diffusion to an Ubuntu image on an *EC2* virtual machine.

diffusion.conf

Ubuntu makes use of the *Upstart daemon* as a replacement for `init`. Copied to `/etc/init/diffusion.conf` this file contains configuration for Upstart to begin Diffusion at boot-time in a background process as a unpriviledged user. It establishes *iptables* rules to route traffic from priviledged ports to Diffusion.

Users may stop and start Diffusion using Upstart commands, e.g. to start the server

```
service diffusion start
```

To stop Diffusion

```
service diffusion stop
```

To check the status of Diffustion

```
service diffusion status
```

In the event that something goes amis Upstart will write a log file to `/var/log/upstart/diffusion.log`

etc/Connectors.xml

Excepting for two port number changes this is an otherwise regular copy of *etc/Connectors.xml* normally found in a Diffusion installation. This edition however binds the Flash policy connector and the Silverlight policy

Copyright 2013 Push Technology

connector to portnumbers greater that 1024, making it possible to run Diffusion as an unprivileged process. This file should be used in conjunction with the iptables rules established in `diffusion.conf`

ec.xml

An illustrative Apache Ant script that can be used to start, stop, and get status from a Diffusion server running on an EC2 Linux host. It also demonstrates an inelegant means of deploying and undeploying a *DAR file* to/from the remote server, by copying the file to the remote server, then moving it into the deploy directory.

Table 61: Targets

Property	Purpose
start	Runs `sudo service diffusion stat` on remote host via SSH
stop	Runs `sudo service diffusion stop` on remote host via SSH
status	Runs `sudo service diffusion status` on remote host via SSH
deploy	Uploads local DAR file to staging dir, then moves it into Diffusion deploy directory
undeploy	Removes DAR file from deploy directory, signalling Diffusion to undeploy related publishers (where possible)

The script is driven by named properties:

Table 62: Properties for targets start, stop and status

Property	Purpose
remote.host	EC2 host running sshd
remote.username	Authentication username, default of `ubuntu`
remote.keyfile	*PEM* encoded key use during authentication

Table 63: Additional properties for targets deploy and undeploy

Property	Purpose
dar.file	Name of a DAR file to deploy or undeploy
remote.diffusion.dir	Root directory of the remote Diffusion installation

Example deployment:

```
ant -Dremote.host=54.235.65.36 \
 -Dremote.keyfile=$HOME/.ssh/ec2-push1.pem \
 -Ddar.file=$HOME/Applications/Diffusion4.5.0/demos/dogfight.dar \
 -Dremote.diffusion.dir=/home/ubuntu/Diffusion/Diffusion4.5.0 \
 deploy
```

The script uses no proprietary Diffusion code and is open to extension during development of a solution. Both the `sshexec` and `scp` tasks depend on the *jsch library* which may have to be downloaded.

Copyright 2013 Push Technology

Tools for Joyent

This is a description of adaptations which are provided for use when using Diffusion™ within the Joyent Cloud.

Diffusion includes in `tools/joyent/` files useful when deploying Diffusion to an SmartOS virtual machine in the *Joyent cloud*

diffusion.xml

SmartOS makes use of *SMF* to start services at boot-time. Once tailored for the host and installed into SMF this file contains configuration for SMF to begin Diffusion at boot-time in a background process as an unprivileged user called `push`. This file does not create or assert the existance of a `push` user.

Users may stop and start Diffusion using SMF commands, e.g. to start the server

```
sudo svcadm enable diffusion
```

To stop Diffusion

```
sudo svcadm disable diffusion
```

Copyright 2013 Push Technology

Chapter 26

Protocol

Topics:

- *DPT*
- *HTTP Protocol*
- *WebSocket Protocol*
- *Command Protocols*

Diffusion™ Protocols - Version 4

In general the Diffusion™ APIs handle the connection and messaging protocols which are therefore hidden from end users, however this section describes the protocols for diagnostic purposes and also to allow use of Diffusion™ from an environment that does not have an API. Please contact Diffusion Support if you wish to make use of the Protocol for a client implementation.

The format

```
0xnn
```

is used throughout this section to represent hexadecimal byte values. e.g.

```
0x21
```

would represent a byte containing a values of hexadecimal 21 (decimal 33).

Overview

The protocols provided by Diffusion are DPT, WebSockets, HTTP Full Duplex, HTTP Chunked Streaming, HTTP XHR, HTTP Iframe and secure versions of all of these protocols.

The DPT protocol stands for Diffusion Protocol over TCP. This is based on creating a TCP connection and using it to send and receive messages in a full duplex way. These connections should be treated by any load balancers as TCP connections.

The WebSocket implementation provides a browser based full duplex connection, built on top of WebSocket framing. This complies with the WebSockets standards and should be usable with any load balancer or proxy with support for websockets.

The HTTP Full Duplex acts like HTTP in the initial connection handshake then acts like DPT for the exchange of messages. Messages are exchanged by the client and the server without using HTTP request/response and load balancers or HTTP proxies are not recommended.

The HTTP Chunked Streaming is also called Comet. It provides a streaming connection for messages from the server by using HTTP chunked encoding. A separate TCP connection is used to send messages from the client to the server. This provides two simplex connections, one based on request/response (upstream) and the other streaming data (downstream). This complies with the HTTP 1.1 standard and load balancers and HTTP 1.1 proxies should be able to cope it you can configure them to permit chuncked encoding and ignore custom headers.

HTTP polling has two flavours, XHR and IFrame. The XHR connection uses additional and custom headers to encode the details of the request. An poll request will remain open until it receives a message. The IFrame uses query string paramters to encode the details of the request. These provide two simplex connections both based on request/response. This complies with the HTTP 1.1 standard and load balancers and HTTP 1.1 proxies should be able to cope it you can configure them to ignore custom headers.

Copyright 2013 Push Technology

DPT

Diffusion™ Protocol over TCP

Special Byte Values

Diffusion™ uses byte based protocols and certain byte values have special meanings. The following representations are used throughout to signify these values.

	Value	Description
MD	0x00	Message Delimiter (Used on for connection)
RD	0x01	Record Delimiter
FD	0x02	Field Delimiter

Connection Protocol

How connections are made to Diffusion™ components over TCP/IP

All connections involve the initiator of the connection sending a 'Connection Request' and receiving a 'Connection Response'.

Connection Request

A connection request has the following format:-

`PVTCcredentials<RD>data<md>`

Where:-

P	Protocol byte - Fixed value of `0x23`
V	Protocol version number as a byte. (8 bit signed two's complement integer) Current version=4
T	The connection type as a byte
C	Specifies the connection capabilities as a byte.
credentials	Specifies optional authentication credentials. These are specified as 'username' and 'password' tokens.For example `username<FD>password` This record only needs to be provided if credential information is required by the server. If no credentials are required then only the `data` can be sent (i.e. no `<RD>` is present). Credentials are always `UTF-8` encoded.
data	The *data* may be one or more fields separated by `<FD>` . Data varies according to connection type. Data is always `UTF-8` encoded.

Connection Response

A connection response has the following format:-

`PVRM data <MD>`

Where:-

P	Protocol byte - Fixed value of # (`0x23`)
V	Protocol version number as a byte (8 bit signed two's complement integer). Current Version=4
R	The connection response as a byte (8 bit signed two's complement integer).
`M`	Message Length Size as a byte (8 bit signed two's complement integer) . This indicates the number of bytes allocated to the Message Size field in a Message. Can have a value of 4, 2 or 1. (See Message Formats).

Copyright 2013 Push Technology

data	The optional response data may be one or more fields (separated by <FD>) and varies according to connection type. Data is always `UTF-8` encoded.

Connection Types and Data Non Client Types

Type	**Connection Data**	**Response Data**	
Event Publisher	01 (`0x01`)		*identifier*
External Publisher	02 (`0x02`)	*publisherName* <FD> *topicSet*<FD>*notifications*	

Notes:

1. ' *identifier* ' is a unique string identifier generated by the Server.
2. ' *publisherName* ' is a unique name for the External Publisher.
3. ' *topicSet* ' is a comma separated list of valid *Topic names* to register.
4. ' *notifications* ' indicates the notifications that the External Publisher wishes to receive.
   This is a list of options separated by " & " from the following list of possibilities -
   "ClientConnected","ClientResolved","ClientUnsubscribed","ClientClosed","ClientInvalidSubscription","ClientQueueThresholds
   The case is not significant.

Client Types

Possible Client Types (in the range 16 to 63) are as follows:-

16 (0x10)	Publisher (Client is a publisher at another server)
20 (0x14)	Default Type (for user use)
21 (0x15)	Java (API)
22 (0x16)	.Net (API)
23 (0x17)	Flash (Plugin)
24 (0x18)	Silverlight (Plugin)
25 (0x19)	iOS
26 (0x1A)	J2ME
27 (0x1B)	Android
28 (0x1C)	Blackberry
29 (`0x1D`)	C Client
30 (`0x1E`)	Perl Client
31 (`0x1F`)	Introspector Client
32 (`0x20`)	Windows Phone Client
33 (`0x21`)	iPad Client

The connection and response data for all Client Types is as follows:-

Connection Data	**Response Data**
topicSet<FD>*client*	*client*

Notes:

Copyright 2013 Push Technology

1. ' *topicSet* ' is a comma separated list of one or more Topic Specifiers indicating the Topics to subscribe to on connection. It can be empty to indicate no Topics.
2. ' *client* ' is a unique string identifier for a Client generated by the Server. It is returned in a Client connection response and may optionally be supplied to a Client connection request to request a connection to a previously established Client session.

Flash Policy Requests

A connector that supports Flash plugin clients may also service policy file requests. In this case what is received would be :-

```
<policy-file-request/>
```

And what would be returned would be a policy file (no response code). No permanent connection would be established.

Response Codes

100 (`0x64`)	OK - Connection Successful
101 (`0x65`)	Invalid Connection Protocol
103 (`0x67`)	One or more of the specified topics are invalid
105 `(0x69)`	Reconnection Successful
110 (`0x6E`)	Topic already exists (External Publisher)
111 (`0x6F`)	Connection Rejected
127 (`0x7F`)	Undefined error

Capabilities

When sending data messages to the Diffusion™ Server the data body may be encoded in one of the accepted manners, in which case the Message encoding should indicate the encoding used. Alternatively, the Message may indicate an encoding that the server should apply to the Message even though the Message is sent to the Server in unencoded form. When using a protocol that can receive Messages from the Server then it is necessary to indicate on connection which encodings can be handled. This is done using the 'capabilities' mask on connection.

This is a mask of one or more of the following values:-

0x01	Can handle encrypted data messages
0x02	Can handle compressed data messages
0x04	Can handle base 64 encoded messages

Thus a value of 0 would indicate that encoded Messages could not be handled and therefore the Server would pass all Messages in unencoded form. A value of 7 would indicate that all byte encodings can be handled and therefore the server can pass encoded Messages. When using DPT this would typically be zero unless one or more of the given encodings can be handled for messages passed from the Server. This value would be ignored for a connection type that does not involve data Messages being passed from the Server.

Messages Protocol

The format of Messages that are sent over connections and the permitted message types for each connection type.

Protocol Version 4

All Diffusion™ Messages are byte based and comprise a fixed header portion, followed by optional header values, followed by an optional data body.

Copyright 2013 Push Technology

The data portion of a Message is optionally encoded with one of a set of permissible byte encodings. These byte encodings are provided for Message compression and encryption.

Character encoding of Strings within the data portion of a Message is handled by the APIs but if users use the raw protocol they need to perform any required character encoding as appropriate. In general it is recommended that characters are encoded using UTF-8 in the data. Where UTF-8 is not used then the Message recipient would need to decode characters as appropriate.

Headers are always UTF-8 encoded.

Message Formats

All messages will take the form:- `LLLLTE{header<RD>}{data}` where:-

`LLLL`	This is the total message length (LLLLTE, headers and data) . Depending upon the system wide configured size of the length portion of a Message, this may comprise 1, 2 or 4 bytes as follows:- L for a 8 bit signed two's complement integer allowing for a maximum message length of 127 bytes. LL for a 16 bit signed two's complement integer allowing for a maximum message length of 32,767 bytes. LLLL for a 32 bit signed two's complement integer allowing for a maximum message length of 2,147,483,647 bytes. The value in use is returned on connection.
`T`	Message type byte.
`E`	Byte encoding byte. This will be zero if there is no data.
`{header <RD>}`	Message headers terminated by a record delimiter. Fixed headers will appear first followed by any user defined headers. The fixed headers vary according to Message type. Where there is more than one header they are separated by <FD> delimiters. Some Message types may have no fixed headers but user headers may be used on any Message type. Messages with no fixed headers and no data do not need the <RD> delimiter.
`{data}`	The Message data (possibly encoded according to the encoding byte). Some Message types may have no Message data.

Byte Encoding Values

0 (`0x00`)	NONE - no encoding applied or requested
1 (`0x01`)	Encryption Requested
2 (`0x02`)	Compression Requested
3 (`0x03`)	Base 64 Encoding Requested
17 (`0x11`)	Encrypted
18 (`0x12`)	Compressed
19 (`0x13`)	Base 64 Encoded

In 'requested' cases the message is not encoded but should be encoded by the server before being passed on to a recipient that has sufficient capabilities.

Message Types

The following table defines all permitted message types.

Copyright 2013 Push Technology

Table 64: Message Types

Type	Description	Fixed Headers (see Key below)	Data
20 (0x14)	Topic Load Message	*topic* (or *topicAlias*)	Y
21 (0x15)	Delta Message	*topic*	Y
22 (0x16)	Subscribe/Register Topic(s)	*topicSet*	N
23 (0x17)	Unsubscribe/ Unregister Topic(s)	*topicSet*	N
24 (0x18)	Ping Server	*timestamp*, *queueSize*	N
25 (0x19)	Ping Client	*timestamp*	N
26 (0x1A)	Credentials	*username*, *password*	N
27 (0x1B)	Credentials Rejected	*username*, *password*	N
28 (0x1C)	Abort Notification		N
29 (0x1D)	Close Request		N
30 (0x1E)	Topic Load - ACK Required	*topic*, *ackId*	Y
31 (0x1F)	Delta - ACK Required	*topic*, *ackId*	Y
32 (0x20)	ACK - acknowledge	*ackId*	N
33 (0x21)	Fetch	*topic* (and optional user headers)	N
34 (0x22)	Fetch Reply	*topic* (and user headers from request if supplied)	Y
35 (0x23)	Topic Status Notification	*topic* (or *topicAlias*), *topicStatus*	N
36 (0x24)	Command Message	*topic*, *command* (, *parameters*)	Y
40 (0x28)	Command Topic Load	*topic*, *commandTopicCategory*, *commandTopicType*	Y
41 (0x29)	Command Topic Notification	*topic*, *notificationType* (, *parameters*)	Y
48 (0x30)	Cancel Fragmented Message Set	correlation id	N

Copyright 2013 Push Technology

Several message types may also be subject to fragmentation. In this case, the message type is bitwise OR'd with `0x40`. Currently, the types Topic Load, Delta and Fetch Reply may be fragmented, leading to the following message types:

84 (`0x54`)	Topic Load - Fragmented	topic, correlation id, fragment number(/total fragments), (total size)	Y
85 (`0x55`)	Delta - Fragmented	topic, correlation id, fragment number(/total fragments), (total size)	Y
98 (`0x62`)	Fetch Reply - Fragmented	topic, correlation id, fragment number(/total fragments), (total size)	Y

Table 65: Key

Term	Description
ackId	A unique Message acknowledgement id - When Client or Server receives a type 30 or 31 it must respond with a type 32 with the same ackId.
command (, *parameters*)	Used on a command topic (i.e. one that uses command topic data) and would be ignored for any other topic. The command is a topic dependent string and the number of parameters is variable.
commandTopicCategory	may have a value of "0" for user or a specific category number that identifies a Diffusion™ type category. See *Command Protocols* for Diffusion implementations. *commandTopicType* is a subType which can be used in any way appropriate to the category. Command Topic Loads can also contain data according to category and type.
notificationType (, *parameters*)	A reply or notification received on a command topic. The type is a topic dependent string and the number of parameters varies by type.
timestamp	A decimal string representation of the number of milliseconds between the current time and midnight, January 1, 1970 UTC.
topic	A single Topic name. Note that where 'Topic Aliases' are in use then all messages from the Server will use the alias (including prefix) rather than Topic name and all Messages to the server may use either Topic name or alias.
topicAlias	An optional value passed on a Topic Load when Topic Aliasing is configured at the Server. It provides an alternative short name and is prefixed by the special alias character (! or 0x21) which may be used in all subsequent message types that need to reference the Topic. Where an alias has been provided the Client must expect all future Messages received for the Topic to contain only the Topic alias.
topicSet	One or more Topic names (not aliases) separated by commas (e.g. A/B,C/D). For subscribe/unsubscribe and Topic Notification Request each entry can be a Topic Selector.
topicStatus	Indicates a change in status of a Topic that the Client was subscribed to. Currently only a value of 'R', meaning Topic removed is implemented.
total fragments, total size	Used on the first message in a fragmented message set.
queueSize	the size of the Client Message queue - zero when part of a ping request.
username/password	authentication credentials as passed on connection but can be used to supply or replace credentials after connection.

Copyright 2013 Push Technology

Message Protocols

The following table defines the permitted message types for each connection type. Assume that any message could have a fragmented counterpart.

Table 66: Message Interactions

	To Server	From Server
Client (All Types)	21 (Send Message) 22 (Subscribe) 23 (Unsubscribe) 24 (Ping Server) 25 (Ping Client Response) 26 (Credentials) 29 (Close Notification) 31 (Send Message - Ack required) 32 (ACK) 33 (Fetch Request) 36 (Command)	20 (Topic Load) 21 (Delta) 24 (Ping Server Response) 25 (Ping Client) 27 (Credentials Rejected) 28 (Abort Notification) 30 (Topic Load - ACK required) 31 (Delta - ACK required) 32 (ACK) 34 (Fetch Reply) 35 (Topic Status) 36 (Command) 40 (Topic Load - Paged) 41 (Delta - Paged) 42 (Cancel Fragmented Message Set)
Event Publisher	20 (Topic Load) 21 (Delta) 29 (Close Notification) 36 (Command)	21 (Delta) 29 (Close Request) 36 (Command)

Fragmented Messages Protocol

When a message is split into fragments, each fragment in the set is assigned the same correlation ID which is transmitted as the first header after the topic name. The first message of a set will also set a header indicating that this is fragment 1 of *n* , where *n* is the total number of fragments which make up this message. Subsequent fragments provide the fragment number in the set. The first fragment will also provide a header indicating the total length of the data once all fragments have been recombined. e.g. For a 250-byte message with a fragment size of 100:

Fragment	Headers
First fragment	topic name correlation ID 1/3 250 {100 bytes of data}
Second fragment	topic name correlation ID

Copyright 2013 Push Technology

	2 {100 bytes of data}
Third fragment	topic name correlation ID 3 {50 bytes of data}

On receipt of a Cancel Fragmented Message Set message (type 42), then the client should expect that no further messages in the set will be received and must discard any fragments with the correlation ID that is passed.

HTTP Protocol

The protocol for connecting over HTTP

Protocol Version 4

As well as the standard socket based protocol it is also possible to perform Client Connections over HTTP Protocol.

Connections can be:-

- Standard HTTP (POST)
- Comet HTTP (POST)
- IFrame (GET)
- Full Duplex over HTTP.

The connection protocol is slightly different from the DPT in that connection parameters and connection responses (with the exception of Full Duplex) are passed in HTTP headers or IFrame parameters.

The message protocol is similar to the DPT except that headers are passed as HTTP headers (except Full Duplex) or IFrame parameters.

Headers/Parameters

m	Method: HTTP Command (See values below)
ty	Type: Identifies the Client Type (See values below).
ca	Capabilities
v	Version: The protocol version - passed on connection and connection response.
username	Credentials user name. (Specified on connection - optional).
password	Credentials password (Specified on connection if username specified).
c	Client ID: Client Identifier - returned on connection (not Duplex)
d	Data: - This forms the Message data body.
t	Topic(s): Optionally on connection and for topic related commands.
u	User Header: (for Messages)
tt	Transport Timeout: (Specified on connection - optional)
s	Sequence: This is the message sequence number. An incrementing sequence number must be sent with each message other than connect and close.
a	Ack ID: This ack ID token when a client send a message where an acknowledgement is required.

Copyright 2013 Push Technology

Method Value

The following are the possible values of the '`method`' parameter or header and is equivalent to a Message type. The values in bold are the same as the corresponding Message type values. Others are HTTP connection specific.

Value	Command Meaning
0	Connect/Register
1	Poll for Messages
2	Send - Message from Client
3	Duplex - Create Duplex Connection
5	SendBatch - Send batched messages from Client (Only for HTTPC)
20	Topic Load
21	Delta
22	Subscribe
23	Unsubscribe
24	Ping Server
25	Ping Client
26	Credentials
27	Credentials Rejected
28	Abort Client
29	Close Client
30	Load Ack
31	Delta Ack
32	Ack
33	Fetch
34	Fetch Reply
35	Topic Status
36	Command
40	Command Load
41	Command Notification
48	Cancel Fragmented Message

Client Types

Possible HTTP Client Types are as follows.The character value is used as the actual `type` header value.

B	Browser Default (Callback)
J	Java (API)
F	Flash
N	.Net

Copyright 2013 Push Technology

S	Silverlight
W	Windows Phone
U	User
WJ	Websocket Java
WB	Websocket Browser
WN	Websocket .NET
CF	Comet Flash
CS	Comet Silverlight
CJ	Comet J2ME
I	Introspector
WI	Websocket Introspector

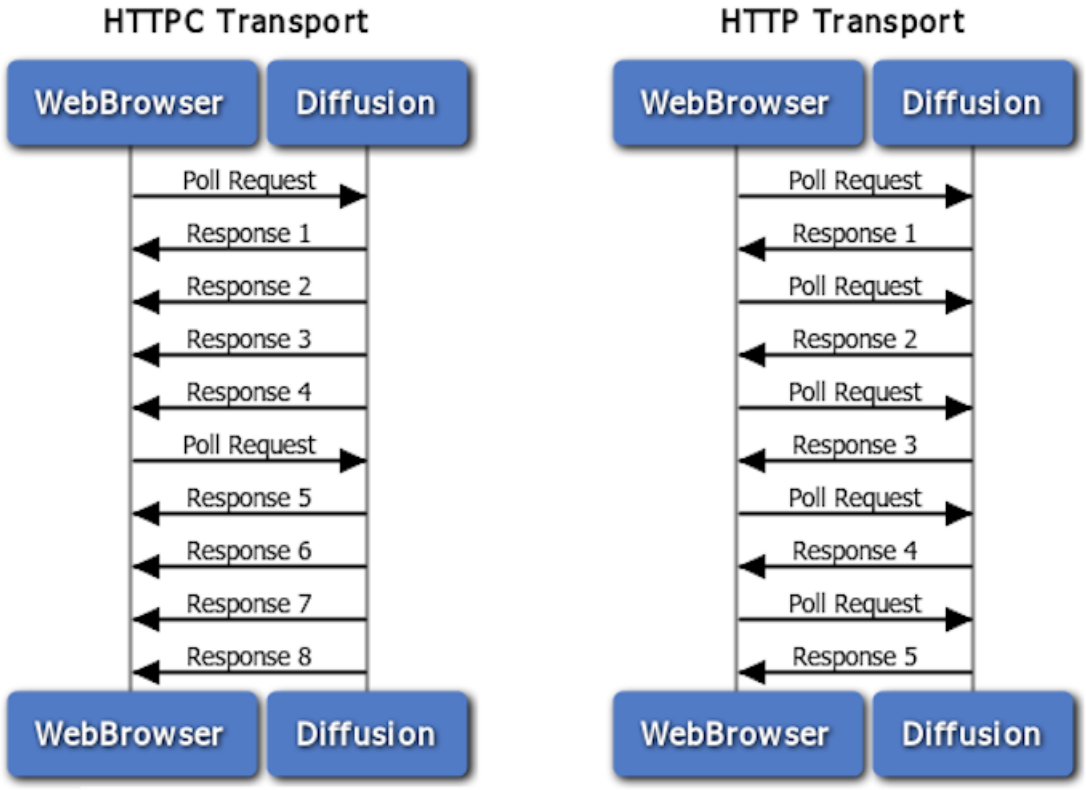
HTTP Comet (HTTPC)

This transport uses long-life HTTP requests and chunked encoding transfer mode as opposed to occasional polling of the server. Instead the server has an open line of communication over which it can push data to the client.

HTTPC can deliver data to the client at any time, not only in response to user input. The data is delivered over a single, previously-opened connection. This approach reduces the latency for data delivery significantly.

The HTTPC request from the client is a polling request that creates a connection to the server. The server can then send messages to the client at any time so long as the connection is open. The server closes the connection after it has sent 30Kb of data (this amount can be configured using the comet-bytes-before-new-poll configuration item, see the *WebServer Client Service configuration section*). The client will then make another poll request. All messages sent to the client will be in response to these poll requests. Messages sent from the client to the server will be sent using POST requests. The message formats are exactly the same as for an HTTP based connection.

Copyright 2013 Push Technology



The HTTPC connection is closed after 30Kb to make it more virus checker friendly. Some virus checkers may hold an HTTPC connection until all the data has been sent through. This may cause batching of messages sent over HTTPC.

You should refer to the *browser limitations* within the client section when using HTTPC. The client libraries we provide need to work through the browser to implement HTTPC.

Full Duplex Protocol

This allows for a full Duplex connection over HTTP. Once a connection is made then the message formats are exactly the same as for a socket based connection, however the data will be sent in HTTP chunked encoding format.

A connection request might therefore take the form:-

```
POST /diffusion/ HTTP/1.1
User-Agent: DiffusionClient
Host: localhost:8080
Transfer-Encoding: chunked
m: 3
ty: U
v: 2
ca: 7
t: Echo
```

The connection response will indicate success or failure in the standard HTTP manner but the HTTP headers will be followed by a connection response buffer of exactly the same format as DPT and then normal message protocol interactions commence.

Copyright 2013 Push Technology

WebSocket Protocol

The protocol for connecting over WebSockets

Protocol Version 4

Diffusion™ has support for accepting connections from current and older versions of the WebSocket protocol including hixie-76. The description of the connection protocol is based on RFC 6455. WebSockets are used to provide full duplex communication over port 80, which should encourage a firewall friendly connection. They are possible over other ports.

WebSockets begin with an HTTP style handshake. It then upgrades to the websocket connection. It then streams data in a different format to DPT, HTTPC or HTTP. The websocket protocol transmits data over the wire in frames. Frames can be sent at any time by both the client and server once the handshake has been completed and before the connection has been closed. Diffusion™ messages may be split over multiple frames. Once the handshake has been completed then the server must send the client ID.

Connection Protocol

To connect using WebSockets the client must make a request to Diffusion™ with the headers "Connection: Upgrade" and "Upgrade: WebSockets". This will identify it as an opening of a websocket connection. The headers Sec-WebSocket-Key and Sec-WebSocket-Version with the appropreate values should also be included as defined in RFC 6455. The information specific to opening a Diffusion™ connection are included in the query string of the HTTP request.

ty	Type: Identifies the Client Type either WB (for WebSockets from browser), WJ (for WebSockets from Java), WN (for WebSockets from .NET).
v	Version: The protocol version - passed on connection and connection response.
username	Credentials user name. (Specified on connection - optional).
password	Credentials password (Specified on connection if username specified).
t	Topic(s): Optionally on connection and for topic related commands.

An example of a websocket connection request from the Drawingboard demo is provided below:

```
GET ws://demo.pushtechnology.com/diffusion?
t=drawingboard&v=4&ty=WB&username=default&password= HTTP/1.1
Origin: http://demo.pushtechnology.com
Host: demo.pushtechnology.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: hVlas/+vGiV3uVuG2rs5Rg==
Sec-WebSocket-Version: 13
```

Diffusion™ will respond with an HTTP response which will include the Sec-WebSocket-Accept header to show that the websocket connection has been established. An example response follows.

```
HTTP/1.1 101 Switching Protocols
Connection: Upgrade
Sec-WebSocket-Accept: 91NYcNxLJPwOylr8mlvB4NoEqDU=
Upgrade: WebSocket
```

This completes a successful WebSocket connection but the connection to Diffusion™ is not yet finished. The server must send through the connection response before it can send or receive messages. The connection response is sent through from the server to the client over the websocket. It is sent as a text frame begining with the Diffusion™

Copyright 2013 Push Technology

protocol version, followed by the string "100" for new connections and "105" for reconnections, followed by the client ID, separated by the field delimiter and terminated by the text postfix byte (0xff).

```
4<FD>10[05]<FD><CLIENT ID>0xff
```

Message Protocol

Messages sent over websockets are not prefixed with the message length, a terminating character is used to end a message. Message encodings are not supported for websockets so no encoding byte is used. A message begins with the message type byte, these are the same values as for the *DPT protocol*. This is followed by the headers, which are terminated by the record delimiter. After this is the payload of the message. The messages ends with the message separator (0x08) or the text type postfix (0xff). Messages may span multiple frames. Diffusion™ messages are sent in data frames using the text opcode.

```
T{header<RD>}{data}0x08
```

Where T is the message type byte.

Command Protocols

Protocol definitions for services that use command messages

The message protocol defines command messages which are automatically routed to and processed by Topics that have CommandTopicData.

There are several implementations of Commad Topics and the message content for each is documented here. This documentatioon is provided for diagnostic purposes only and it is not expected that client applications would need to know these message formats as the Diffusion Client APIs should provide appropriate handlers which will format and unpck the messages.

Service Topic Protocol

The Command protocol used for Service Topics

A 'Service Topic' is a type of command topic used for request/response type services. Client APIs provide handlers for this protocol.

Topic Load (Message Type 40)

The *commandTopicCategory* of the Topic Load will be "0".

The *commandTopicType* will be a "Service Type' as defined for the Topic Data.

The headers and data can be anything set on the Topic Data.

Commands (Message Type 36)

Type	Description	Headers / Data
requestType	The request type can be anything that the service would understand.	0 = Request Id This is a unique correlation id generated by the client and will be returned with any response. Further headers and data may be supplied as input to the request.

Notifications (Message Type 41)

Type	Description	Headers / Data
responseType	The response type can be anything as defined by the service.	0 = Request Id 1-n = Other user headers (optional)

Copyright 2013 Push Technology

		Data = optional user data
!ERR	Error response	0 = Request Id 1 = Error Type SERVICE("SRV"), DUPLICATE("DUP"), TIMEOUT("TIM"), INVALID("INV"), USER("USR"); Data is in fields as follows:- 0 = Error Message 1 = Exception message (optional) 2 = Further info (optional)

Paged Topic Protocol

Defines the usage of the Command protocol for Paged Topic Data

A 'Paged Topic' is a type of command topic. Client APIs provide handlers for this protocol.

Topic Load (Message Type 40)

The commandTopicCategory of the Topic Load will be "1".

The commandTopicType may be:- "PS" for Paged String Data "PR" for Paged Record Data

At future releases the data body may contain metadata.

Commands (Message Type 36)

Type	Description	Headers / Data
O	Open Paging	0 = Lines per page 1 = Start page (-1 for last)
R	Refresh Page	
N	Next Page	
P	Prior Page	
F	First Page	
L	Last Page	
n (a numeric)	Page number	
C	Close Paging	

Notifications (Message Type 41)

Type	Description	Headers / Data
P	Page Delivered in response to O,R,N,P,F,L commands.	0 = Current Page Number 1 = Last Page Number 2 = Total Number of Lines Data = lines of data separated according to type.

Copyright 2013 Push Technology

S	Status Update to status so page range or dirty mark can be refreshed.	0 = Current Page Number 1 = Last Page Number 2 = Total Number of Lines 3 = dirty (true or false)
U	Update A line on the current page has been updated.	0 = Current Page Number 1 = Last Page Number 2 = Total Number of Lines 3 = Index (on page) of line to update Data = single line of data
A	Add Lines have been added to the end of the current page	0 = Current Page Number 1 = Last Page Number 2 = Total Number of Lines Data = lines of data (will not be more than fills the page).

Notification Topic Protocol

The usage of Command message protocol by Topic Notification Topic Data

This specification defines the use of Version 2 (available at Diffusion release 4.6) of the Notification Topic Protocol.

A 'Topic Notification Topic' is a type of command topic. Client APIs may provide handlers for this protocol.

Topic Load (Message Type 40)

The commandTopicCategory of the Topic Load will be "`3`".

The commandTopicType will be set to "`TN`"

Header 0 will contain the protocol version number (currently `2`)

The usage of the body is to be defined

Commands (Message Type 36)

Table 67: Commands (Message Type 36)

Type	Description	Headers / Data
`S`	Select/Amend Notifications	H0 = Mode - `R`=Replace,`A`=Append,`D`=Remove,`X`=Clear H1 = TopicSet (comma separated selectors)
`L`	Set Detail Level	H0 = Level `0` = Minimum (only type) - this is the default `1` = Properties only `2` = Metadata only `3` = All `4` = None - special case if just deletions and/or updates required H1 = Metadata Caching (`true` or `false`) H2 = Notify Deletions (`true` or `false`) - false if not present H3 = Notify Updates (`true` or `false`) - false if not present

Copyright 2013 Push Technology

Table 68: Notifications (Message Type 41)

Type	Description	Headers / Data
A	Topic Added	H0 = Topic Name data:- (3 records - if not selected by level the record will be there but empty) R0 Basic info F0=Topic Type(see below) R1 Topic Properties key/value field pairs R2 Metadata F0=Metadata type X=XML,R=Reference(name), N=None F1=Metadata reference (type X and R only) F2=Metadata content (type X only)
D	Topic Removed	H0 = Topic Name
U	Topic Properties Changed	R1 Topic Properties key/value field pairs

Table 69: Topic Types

Code	Type	Metadata type
N	Simple	
S	Single Value	MField
R	Record	MMessage
G	Protocol Buffers	optional any
U	Custom	optional any
SD	Slave	from target
SV	Service	optional any
PS	Paged String	
PR	Paged Record	MRecord
TN	Topic Notification	
RO	Routing	
C	Child List	
RC	Remote Control Edge	
RS	Remote Service Edge	
RCR	Remote Control Relay	
RSR	Remote Service Relay	

Table 70: Topic Properties

Key	Value	Type	Topic Types	Optional/Mandatory & default
ATC	Attachment class name	String	all	Optional - default no attachment

Copyright 2013 Push Technology

Key	Value	Type	Topic Types	Optional/Mandatory & default
LOC	Lockable	Boolean	all	Optional - default true
REF	Reference	String	all	Optional - default no reference
SHC	TopicSubscriptionHandler class name	String	all	Optional - default no handler
FHC	TopicFetchHandler class name	String	all	Optional - default no handler
DMC	Delta Message Capacity default	Integer	all	Optional - default -1
LMC	Load Message Capacity default	Integer	all	Optional - default -1
ISM	Introspection Mask	Byte	all	Optional - default 0
LTO	Lock Timeout	Long	all	Optional - default 5000
TOU	Tidy on Unsubscribe	Boolean	all	Optional - default false
ALI	Aliasing	Boolean	all	Optional - default as Publisher
DTN	Domain Topic Name	String	*sub*	Optional
LEN	Load Encoding	Byte	all	Optional - default 0
DEN	Delta Encoding	Byte	all	Optional - default 0
LFS	Load Fragment Size	Integer	all	Optional - default no fragments
DFS	Delta Fragment Size	Integer	all	Optional - default no fragments
LFL	Load Fragment Lifecycle (S=n,M=n,P=n)	String	all	Optional - default no lifecycle
DFL	Delta Fragment Lifecycle (S=n,M=n,P=n)	String	all	Optional - default no lifecycle
DIC	TopicDataInitialiser class name	String	*pub*	Optional - default no initialiser
LHD	Load Headers (field delimited list)	List	*pub*	Optional - default no load headers
LAR	Load ack required	Boolean	*pub*	Optional - default no ack required
DAR	Delta ack required	Boolean	*pub*	Optional - default no ack required
G_PC	Proto Class Name	String	G	Mandatory for type
G_MN	PB Message Name	String	G	Mandatory for type
G_UM	PB Update Mode - (P)artial or (F)ull	String	G	Optional - default P
G_DV	PB Deletion Value	String	G	Optional - default "<DEL>"
U_HC	CustomTopicDataHandler class name	String	U	Mandatory for type
R_EFV	Empty Field Value	String	R	Optional - default = ""
PX_CC	Paged Comparator Class	String	PR,PS	Optional - default unordered
PX_DP	Paged Duplicates Policy - (N)ot allowed, (F)irst or (L)ast	String	PR,PS	Optional - Default 'N'
SV_TY	Service Type	String	SV	Mandatory for type
SV_HC	ServiceHandler class name	String	SV	Mandatory for type
SV_TO	Service Request Timeout	Long	SV	Optional - default 5000
SV_TT	Service Target Topic	String	SV	Optional - default parent is used

Copyright 2013 Push Technology

Key	Value	Type	Topic Types	Optional/Mandatory & default
SV_HD	Service Header Options	List	SV	Optional - default no options
SD_MT	Slave master Topic	String	SD	Mandatory for type
RO_HC	RoutingTopicDataSubscriptionHandler	String	RO	Mandatory for type
TN_MC	Notify - Metadata caching	Boolean	TN	Optional - default false
TN_NA	Notify - Notify all topics	Boolean	TN	Optional - default false

pub refers to Publishing Topics - e.g. U,G,R,S,SD

sub refers to any Topics that can be subdomain Topics under a Remote Control domain Topic.

'List' type is a list of Strings delimited by list delimiter byte (x06).

Copyright 2013 Push Technology